Semantic Classical Music REST API

Tiago Gomes
up201806658@up.pt

João Matos
up201703884@up.pt

João Sousa
up201806613@up.pt

December 12, 2022

Abstract

The technological progress of recent decades has brought new data-related concepts to the spotlight, such as the Semantic Web [1] and Linked Data [2], and new standards and technologies, such as RDF (Resource Description Framework) [3], SPARQL (SPARQL Protocol and RDF Query Language) [4], and OWL (Web Ontology Language) [5], in an effort to make the information on the Web machine-readable, and by consequence, more valuable.

In this work, and within the scope of the Semantic Web and Linked Data, by connecting already existing data sources (DBTune Classical [6] and DBpedia [7]), we propose a classical music API where users with no knowledge in query languages, can have information needs (that go from compositions made by a certain composer during a certain year, to the list of composers that influenced or were influenced by another) fulfilled; some usage examples are presented, in the form of queries.

1 Introduction

The World Wide Web was invented in 1989 by Sir Tim Berners-Lee [8]. One of its main characteristics was the hyperlink, connecting documents and pages despite their physical location and therefore providing an abstraction to the different layers that were part of the data exchanges. The Semantic Web also represents an abstraction of the information exchange on the Web with a slight but significant difference: instead of only connecting documents, it makes it possible to link to a specific resource within a document.

The term Semantic Web refers to the vision of a Web of Linked Data of the World Wide Web Consortium (W3C) [1], being an extension of the World Wide Web whose goal is to shape it into a more connected network of data, enabling meaning that is at the same time machine and human-readable. For this purpose, some technologies and standards were defined, some of them: RDF (Resource Description Framework), a data modeling language, SPARQL (SPARQL Protocol and RDF Query Language), and OWL (Web Ontology Language).

In turn, Linked Data is the set of design principles that, when obeyed, allow this sharing of data on the Web. In 2010 Sir Tim Berners-Lee proposed a 5-star deployment scheme for Linked Open Data. From 1-star to 5-stars, the data must be (cumulatively): available on the web, structured and machine-readable, must not require proprietary software to analyze it, must use open standards from W3C (e.g., RDF, SPARQL), and must allow its users to discover more relationships within their linked data [2].

2 Requirements & Goals

This work has as its primary goal the implementation of a REST API [9] for Classical Music within the Semantic Web and Linked Data context. To achieve that, some requirements were specified. As the end product needs to allow exploration of the data present in our knowledge graph, queries implemented in our API are an adequate and accessible solution so that the users can obtain new information without the need to understand SPARQL (the standard query language and protocol for Linked Open Data and RDF databases). With these queries, users would also perform CRUD (Create, Read, Update, Delete) [10] operations on each of the types of resources available (them being composer, conductor, event, and musical work). The other requirement proposed was using federated queries, i.e., the integration/linking of distributed RDF data, improving the information provided to the end user.

3 Existing Solutions

Currently, some available working alternatives provide linked data related to music. Yet, despite being close to what is envisioned as the result of this work, they are not focused on classical music and/or require a basic level of understanding of query languages (SPARQL). Some examples are Musicbrainz [11] and the Classical Music Navigator (DBTune) [12]. The first one comprehends a broader scope of music. In contrast, the second one is centered on classical music, but both present levels of knowledge that are more superficial than what we propose (which focuses solely on classical music and tries to increase the reach of the user queries).

4 Knowledge Sources

In this work, the knowledge sources that were incorporated were DBTune and DBpedia. When it comes to ontologies, it is to highlight the DBpedia Ontology [13], a cross-domain and shallow ontology based on the Wikipedia [14] infoboxes, and the Music Ontology [15] that is extensively utilized in DBTune, which is based on RDF and built on top of four main ontologies - FOAF (entity description) [16], Event Ontology [17], Timeline Ontology [18], and FRBR ontology (vocabulary for the description of works, expressions, manifestations, among others) [19] -, allowing future extensibility and easier linking with other resources.

4.1 DBTune

Starting with the most relevant for the classical music theme, DBTune itself is not a dataset but a host for several services, datasets, and resources concerning music.

The ones most pertinent for this work are the Classical Music Navigator dataset, a collection of around 10 000 triples that describes classical composers and their influence network, DBTune classical data, which allows the inter-linking of a variety of contents related to the canon of Western Classical Music with resources from DBpedia via the `owl:sameAs` property, and Musicbrainz, an open source music encyclopedia which has its data available as RDF linked data.

Some other examples that less relevant to the goals of this work are Jamendo (a repository of Creative Commons licensed music) [20], Magnatune (an independent music label) [21], Chord symbol service (an integral part of the chord ontology that can provide chords descriptions in RDF format) [22], and Echonest Analyze [23], an XML to Music Ontology [15] (and Audio Features Ontology [24]) RDF translation service.

4.2 DBpedia

DBpedia is a community-driven project that aims to extract structured information from Wikipedia and make it available on the web using Semantic Web and Linked Data technologies. This information is freely accessible to anyone.

Existing mappings take the structured information of Wikipedia (e.g., infoboxes and tables) and represent it in the DBpedia ontology, consisting of hundreds of classes described by 1650 different properties.

DBpedia is a large-scale project, covering many different topics across dozens of languages concerning the other Wikipedia editions, that can be utilized in many application domains from expressive query answering to topic recognition, and allows linking of datasets on the Web to Wikipedia data, functioning as a knowledge hub.

5 Architecture

The architecture of the REST API is illustrated in Figure 1, present in the appendix. The application runs on a docker [25] environment with two containers, *fuseki* and *middleware*.

The first step is to download the dataset RDF file from the DBTune classical website, which has a SPARQL endpoint available at `http:`

//dbtune.org/classical/sparql. When the *middleware* container is executed for the first time, the RDF file is fed to the application.

The *middleware* container is composed of two components: the Java Spring Boot [26], which is the REST API available in port 8080, and the Apache Jena [27], an element that makes the connection between the API and the triplestore.

The *fuseki* container contains the Apache Jena Fuseki [28], the application triplestore, and SPARQL server. The container exposes the application SPARQL endpoint in port 3030, where is also available the Fuseki UI. The Fuseki UI is a web application where it is possible to execute SPARQL queries on the triplestore, and it is very helpful for testing purposes and/or for more advanced users.

Finally, DBpedia, which has a SPARQL endpoint available at https://dbpedia.org/sparql, is a crucial component when executing SPARQL federation, a feature elaborated in section 6.3.

6 Implemented Queries

6.1 CRUD

One of the original goals of this project was to create endpoints that allowed a user to perform basic CRUD (Create, Read, Update, Delete) operations with each type of resource available in the dataset (composers, conductors, events, and musical works). However, implementing the update functionality turned out to be more difficult than expected. As a result, this was implemented as a deletion followed by the creation of the same resource with new data. Because this is suboptimal, this functionality is only offered for musical works. Table 1 summarizes the information related to endpoints that were created for this section.

6.2 Non-federated queries

More complex queries that could be useful to a user were also implemented. These retrieve more specific information than the ones used in the previous section. The data retrieved by each query is listed below.

- The musical works of a given composer
- The musical works that contain a certain (musical) key
- The composers who influenced a given composer
- The composers who were influenced by a given composer
- The parts of a given musical work
- The compositions from a given year
- The compositions from a period between two given years
- The compositions from a given location

Table 2 summarizes the information related to endpoints created for this section.

6.3 Federated queries

SPARQL federation is a way of querying multiple SPARQL endpoints simultaneously. This allows a user to access and combine data from different sources in a single query, effectively treating the distributed data sources as if they were a single virtual database. In our application, we used SPARQL federation to combine information from our dataset with DBpedia.

The dataset retrieved from DBTune Classical contains triples using the *owl:sameAs* predicate for the majority of the composers, stating that a composer resource in our dataset it's the same as the composer resource in other knowledge bases, namely DBpedia.

To store the necessary metadata to use SPARQL federation, such as the SPARQL endpoints of the external datasets, we used the void vocabulary [29]. This vocabulary describes an RDF dataset using the Resource Description Framework (RDF) data model. It defines a set of terms and properties that can provide metadata about an RDF dataset, such as its name, the entities or concepts described in the dataset, and the URLs of other related datasets. This metadata is stored in an RDF file called void.ttl, available in Listing 27, which, in addition to storing metadata about the external datasets, can be used to help other users or applications discover and understand our dataset.

Table 3 summarizes the information related to endpoints created for this section.

The endpoint implemented on *composer/dbpedia-federation/<composer_id>* takes the id of the composer and retrieves all its information available on DBpedia, namely the predicate and objects of all its linked data, as well as its English labels, when available. Note that the *DBpedia: void:sparqlEndpoint ?sparqlEndpoint* triple pattern is used to get the DBpedia SPARQL endpoint from the metadata fed into our knowledge graph using the void.ttl file. Then, the *?sparqlEndpoint* variable is used after the *SERVICE* keyword to reach the DBpedia knowledge base. The *OPTIONAL* keyword is used to ensure the query does not fail when the predicate or object English label does not exist.

Finally, the query from Listing 26 takes a composer's id and retrieves its birth and death places, as well as the respective coordinates, from DBpedia. This query is just an example of the potential SPARQL federation, as it takes advantage of a massive dataset from the Linked Open Data, allowing users to access previously unavailable information.

7 Current Limitations & Future Work

As mentioned in section 6.1, one of the limitations of the REST API is to make update requests on the resources, except for the musical works. We decided not to implement this feature, as it is not as important as others, but it's on our plans to implement it in future developments.

Another limitation is the fact that the range of the implemented queries is relatively short. The queries implemented in our application aim to be an example and show its potential. More complex queries can be implemented in the future.

Finally, the application would be more user-friendly if it had a frontend component. Having the REST API implemented to support it, a frontend, is a fundamental component to achieving our goal of spreading classical musical information to more users, so it's definitely on our plans to implement it in future works.

8 Conclusion

In this work, we proposed a classical music API for the Semantic Web and Linked Data context. Our API allows users to explore data within a knowledge graph using queries, and it also allows CRUD operations on the available resources. We demonstrated the use of our API through several examples, and we showed how it could be used to answer questions about classical music without requiring any knowledge of query languages. Additionally, we used federated queries to integrate data from multiple sources, improving the information provided to the end user. Our API represents a valuable tool for users who want to learn more about classical music and its connections to other composers and works.

8.1 Ranking based on Open Data principles

Based on the Linked Open Data principles, the project, technically, would not even get the first star (it is not available online due to the costs associated with hosting). However, the requirements for the other four stars are met:

- The data is in a format that is readable by machines
- The format is not proprietary
- W3C standards are used
- There are outgoing links to other datasets (DBpedia)

References

[1] Semantic Web - W3C. https://www.w3.org/standards/semanticweb/.

[2] Tom Heath e Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on Data, Semantics, and Knowledge. Springer International Publishing, Cham, 2011.

[3] RDF - Semantic Web Standards. https://www.w3.org/RDF/.

[4] SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/.

[5] OWL - Semantic Web Standards. https://www.w3.org/OWL/.

[6] DBTune - Classical RDF Service. http://dbtune.org/classical/.

[7] DBpedia. https://www.dbpedia.org/.

[8] World Wide Web. *Wikipedia*, novembro 2022.

[9] Rest apis. https://www.ibm.com/cloud/learn/rest-apis, maio 2021.

[10] shirhatti. CRUD (Create, Read, Update, Delete). https://learn.microsoft.com/en-us/iis-administration/api/crud.

[11] MusicBrainz - The Open Music Encyclopedia. https://musicbrainz.org/.

[12] Classical Music Navigator - RDF. http://dbtune.org/cmn/.

[13] DBpedia Ontology. https://www.dbpedia.org/resources/ontology/.

[14] Wikipedia. https://www.wikipedia.org/.

[15] The Music Ontology. http://musicontology.com/.

[16] FOAF Vocabulary Specification. http://xmlns.com/foaf/0.1/.

[17] Event Ontology. http://150.146.207.114/lode/extract?url=https://raw.githubusercontent.com/luigi-asprino/MON/main/src/event.owl.

[18] Timeline Ontology. https://motools.sourceforge.net/timeline/timeline.html.

[19] Expression of Core FRBR Concepts in RDF. https://vocab.org/frbr/core.html.

[20] Jamendo Music | Free music downloads. https://www.jamendo.com/.

[21] Magnatune: A music service that isn't evil. http://magnatune.com/.

[22] Chord Ontology Specification. https://motools.sourceforge.net/chord_draft_1/chord.html#symbolservice.

[23] DBTune - Echonest Analyze XML to Music Ontology RDF. http://dbtune.org/echonest/.

[24] Alo Allik. The Audio Feature Ontology. Zenodo, junho 2016.

[25] Docker: Accelerated, Containerized Application Development. https://www.docker.com/, maio 2022.

[26] Spring Boot. https://spring.io/projects/spring-boot.

[27] Apache Jena - Home. https://jena.apache.org/.

[28] Apache Jena - Apache Jena Fuseki. https://jena.apache.org/documentation/fuseki2/.

[29] Void vocabulary. https://www.w3.org/TR/void/.

9 Appendix

9.1 Architecture

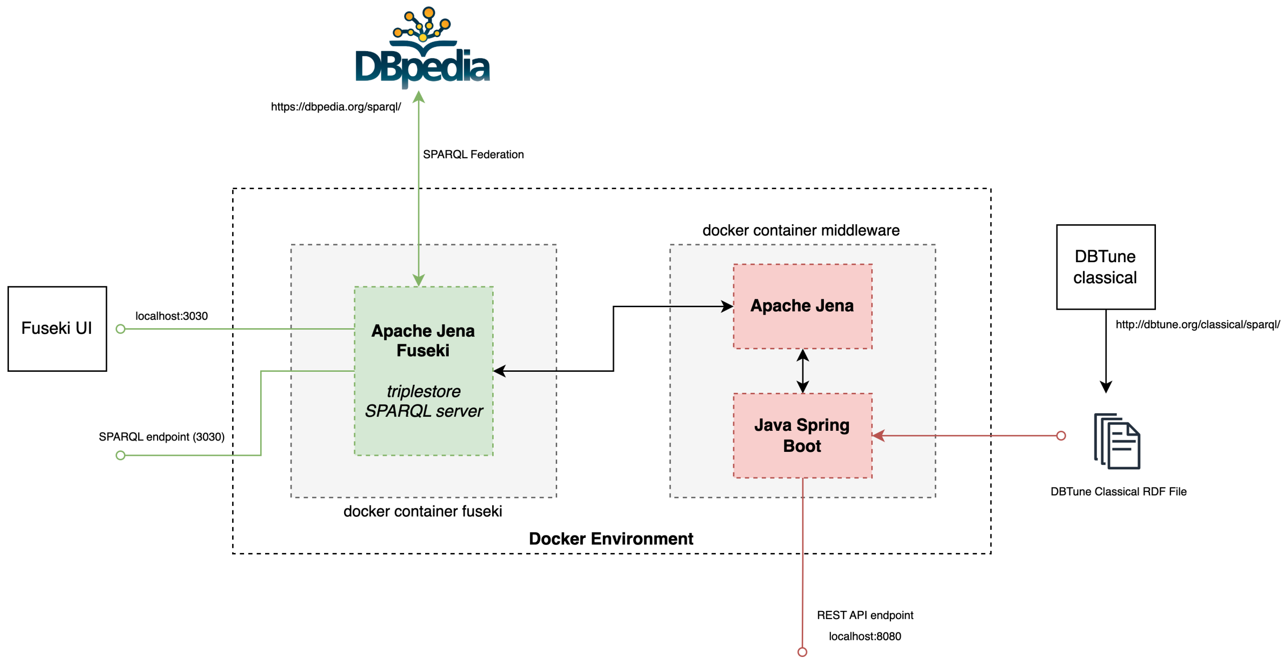

Figure 1: *Architecture*

9.2 Endpoints

9.2.1 CRUD and Search Endpoints

HTTP Method	Path	Input Data	SPARQL Query
GET	event/<event_id>	The id of the event.	See Listing 10.
DELETE	event/<event_id>	The id of the event.	See Listing 11.
POST	event	URI and triples of the event.	See Listing 12.
GET	event/search/<query>	The search query.	See Listing 13.
GET	composer/<composer_id>	The id of the composer.	See Listing 1.
DELETE	composer/<composer_id>	The id of the composer.	See Listing 3.
POST	composer	URI and triples of the composer.	See Listing 4.
GET	composer/search/<query>	The search query.	See Listing 5.
GET	work/<composer_id>/<work_id>	The id of the work and the composer.	See Listing 14.
DELETE	work/<composer_id>/<work_id>	The id of the work and the composer.	See Listing 15.
POST	work	URI and triples of the work.	See Listing 16.
PUT	work/<composer_id>/<work_id>	The id of the work and the composer, the triples of the work.	See Listings 15 and 16.
GET	work/search/<query>	The search query.	See Listing 17.
GET	conductor/<conductor_id>	The id of the conductor.	See Listing 6.
DELETE	conductor/<conductor_id>	The id of the conductor.	See Listing 7.
POST	conductor	URI and triples of the conductor.	See Listing 8.
GET	conductor/search/<query>	The search query.	See Listing 9.

Table 1: *CRUD and Search Endpoints*

9.2.2 Non-federated

HTTP Method	Path	Input Data	SPARQL Query
GET	queries/composerWorks?composerId=<composer_id>	The id of the composer.	See Listing 18.
GET	queries/workKeys?key=<key>	The desired key.	See Listing 19.
GET	queries/composersWhoInfluenced?composerId=<composer_id>	The id of the composer.	See Listing 20.
GET	queries/composersWhoWereInfluenced?composerId=<composer_id>	The id of the composer.	See Listing 21.
GET	queries/partsOfWork?composerId=<composer_id>&workId=<work_id>	The id of the composer and the work.	See Listing 22.
GET	queries/compositionsByYear?year=<year>	The desired year.	See Listing 23.
GET	queries/compositionsByTimeRange?year1=<year1>&year2=<year2>	The upper and lower bounds of the range.	See Listing 24.
GET	queries/compositionsByPlace?place=<place>	The desired place.	See Listing 25.

Table 2: *Non-federated Endpoints*

9.2.3 Federated

HTTP Method	Path	Input Data	SPARQL Query
GET	composer/dbpedia-federation/<composer_id>	The id of the composer.	See Listing 2.
GET	queries/composerLocations?composerId=<composer_id>	The id of the composer.	See Listing 26.

Table 3: *Federated Endpoints*

9.3 SPARQL Queries

9.3.1 Composers

```
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?composer ?predicate ?object
WHERE { ?composer rdf:type type:Composer ;
     ?predicate ?object .
FILTER (
REGEX(
STR( ?composer ),
"^http://dbtune.org/classical/resource/composer/beethoven_ludwig_van$"
) ) }
```

Listing 1: *SPARQL query to get composers*

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX void: <http://rdfs.org/ns/void#>
PREFIX DBpedia: <https://www.dbpedia.org/>

SELECT DISTINCT ?predicate ?predicateLabel ?value ?valueLabel
WHERE {
  DBpedia: void:sparqlEndpoint ?sparqlEndpoint .
  <http://dbtune.org/classical/resource/composer/beethoven_ludwig_van>
    owl:sameAs ?externalResource .
  filter ( regex(str(?externalResource), "dbpedia")) .

  SERVICE ?sparqlEndpoint {
   ?externalResource ?predicate ?value .

  OPTIONAL {
          ?predicate rdfs:label ?predicateLabel .
   FILTER (lang(?predicateLabel) = "en") .
  }

  OPTIONAL {
          ?value rdfs:label ?valueLabel .
   FILTER (lang(?valueLabel) = "en") .
  }

 FILTER ( IF (isLiteral(?value), lang(?value) = "en", TRUE) ) .
 }
}
```

Listing 2: *Federated SPARQL query to get composers*

```
DELETE { ?composer ?predicate ?object . }
WHERE { ?composer ?predicate ?object .
FILTER ( ( REGEX( STR( ?composer ), "^http://dbtune.org/classical/
   resource/composer/beethoven_ludwig_van$" ) || REGEX( STR( ?object )
   , "^http://dbtune.org/classical/resource/composer/
   beethoven_ludwig_van$" ) ) ) }
```

Listing 3: *SPARQL query to delete composers*

```
INSERT DATA { <http://dbtune.org/classical/resource/composer/
   beethoven_ludwig_van> <http://www.w3.org/1999/02/22-rdf-syntax-ns#
   type> <http://dbtune.org/classical/resource/type/Composer> . }
```

Listing 4: *SPARQL query to create composers*

```
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?composer ?predicate ?object
WHERE { ?composer rdf:type type:Composer ;
    ?predicate ?object .
FILTER ( REGEX( STR( ?object ), "beethoven", "i" ) ) }
```

Listing 5: *SPARQL query to search for composers*

9.3.2 Conductors

```
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?conductor ?predicate ?object
WHERE { ?conductor rdf:type type:Conductor ;
    ?predicate ?object .
FILTER ( REGEX( STR( ?conductor ), "^http://dbtune.org/classical/
   resource/conductor/alain_trudel$" ) ) }
```

Listing 6: *SPARQL query to get conductors*

```
DELETE { ?conductor ?predicate ?object . }
WHERE { ?conductor ?predicate ?object .
FILTER ( ( REGEX( STR( ?conductor ), "^http://dbtune.org/classical/
   resource/conductor/alain_trudel$" ) || REGEX( STR( ?object ), "^
   http://dbtune.org/classical/resource/conductor/alain_trudel$" ) ) )
    }
```

Listing 7: *SPARQL query to delete conductors*

```
INSERT DATA { <http://dbtune.org/classical/resource/conductor/
   alain_trudel> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <
   http://dbtune.org/classical/resource/type/Conductor> . }
INSERT DATA { <http://dbtune.org/classical/resource/conductor/
   alain_trudel> <http://www.w3.org/2002/07/owl#sameAs> <http://
   dbpedia.org/resource/Alain_Trudel> . }
```

Listing 8: *SPARQL query to create conductors*

```
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?conductor ?predicate ?object
WHERE { ?conductor rdf:type type:Conductor ;
    ?predicate ?object .
FILTER ( REGEX( STR( ?object ), "charles", "i" ) ) }
```

Listing 9: *SPARQL query to search for conductors*

9.3.3 Events

```
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?event ?predicate ?object
WHERE { ?event ?predicate ?object .
FILTER ( REGEX( STR( ?event ), "^http://dbtune.org/classical/resource/
   event/9e374152135b$" ) ) }
```

Listing 10: *SPARQL query to get events*

```
DELETE { ?event ?predicate ?object . }
WHERE { ?event ?predicate ?object .
FILTER ( ( REGEX( STR( ?event ), "^http://dbtune.org/classical/
   resource/event/123$" ) || REGEX( STR( ?object ), "^http://dbtune.
   org/classical/resource/event/123$" ) ) ) }
```

Listing 11: *SPARQL query to delete events*

```
INSERT DATA { <http://dbtune.org/classical/resource/event/123> <http
   ://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://purl.org/vocab
   /bio/0.1/Birth> . }
```

Listing 12: *SPARQL query to create events*

```
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?event ?predicate ?object
WHERE { ?event ?predicate ?object .
FILTER ( ( REGEX( STR( ?event ), "^http://dbtune.org/classical/resource/event/
```

Listing 13: *SPARQL query to search for events*

9.3.4 Works

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX ns2: <http://purl.org/ontology/mo/>
SELECT DISTINCT ?work ?predicate ?object
WHERE { ?work rdf:type ns2:MusicalWork ;
```

```
    ?predicate ?object .
FILTER ( REGEX( STR( ?work ), "^http://dbtune.org/classical/resource/
   work/beethoven_ludwig_van/WoO_7$" ) ) }
```

Listing 14: *SPARQL query to get works*

```
DELETE { ?work ?predicate ?object . }
WHERE { ?work ?predicate ?object .
FILTER ( ( REGEX( STR( ?work ), "^http://dbtune.org/classical/resource
   /work/beethoven_ludwig_van/WoO_7$" ) || REGEX( STR( ?object ), "^
   http://dbtune.org/classical/resource/work/beethoven_ludwig_van/
   WoO_7$" ) ) ) }
```

Listing 15: *SPARQL query to delete works*

```
INSERT DATA { <http://dbtune.org/classical/resource/work/
   beethoven_ludwig_van/WoO_7> <http://www.w3.org/1999/02/22-rdf-
   syntax-ns#type> <http://purl.org/ontology/mo/MusicalWork> . }
```

Listing 16: *SPARQL query to create works*

```
PREFIX ns2: <http://purl.org/ontology/mo/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?work ?predicate ?object
WHERE { ?work rdf:type ns2:MusicalWork ;
    ?predicate ?object .
FILTER ( REGEX( STR( ?object ), "moonlight", "i" ) ) }
```

Listing 17: *SPARQL query to search for works*

9.3.5 Other queries

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX ns2: <http://purl.org/ontology/mo/>
SELECT DISTINCT ?work ?predicate ?object
WHERE { ?work rdf:type ns2:MusicalWork ;
    ?predicate ?object .
FILTER ( REGEX( STR( ?work ), "^http://dbtune.org/classical/resource/
   work/beethoven_ludwig_van/" ) ) }
```

Listing 18: *SPARQL query to get a composer's works*

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX ns2: <http://purl.org/ontology/mo/>
SELECT ?work
WHERE { ?work rdf:type ns2:MusicalWork ;
    ns2:key "F␣minor" . }
```

Listing 19: *SPARQL query to get works by key*

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX ns2: <http://purl.org/ontology/mo/>
PREFIX ns4: <http://purl.org/ontology/classicalmusicnav#>
SELECT ?composer
WHERE { ?composer rdf:type type:Composer ;
    ns4:hasInfluenced ?influencedComposer .
FILTER ( REGEX( STR( ?influencedComposer ), "^http://dbtune.org/
   classical/resource/composer/beethoven_ludwig_van$" ) ) }
```

Listing 20: *SPARQL query to get the composers that influenced a given composer*

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX ns2: <http://purl.org/ontology/mo/>
PREFIX ns4: <http://purl.org/ontology/classicalmusicnav#>
SELECT ?composer
WHERE { ?composer rdf:type type:Composer ;
    ns4:influencedBy ?composerWhoInfluenced .
FILTER ( REGEX( STR( ?composerWhoInfluenced ), "^http://dbtune.org/
   classical/resource/composer/beethoven_ludwig_van$" ) ) }
```

Listing 21: *SPARQL query to get the composers that were influenced by a given composer*

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX ns2: <http://purl.org/ontology/mo/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?part
WHERE { <http://dbtune.org/classical/resource/work/purcell_henry/Z_630
   > dc:hasPart ?part . }
```

Listing 22: *SPARQL query to get the parts of a work*

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX ns2: <http://purl.org/ontology/mo/>
PREFIX ns5: <http://purl.org/vocab/bio/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?composition
WHERE { ?composition rdf:type ns2:Composition ;
    ns5:date ?date .
FILTER ( STR( ?date ) = "1883" ) }
```

Listing 23: *SPARQL query to get compositions from a given year*

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX ns2: <http://purl.org/ontology/mo/>
PREFIX ns5: <http://purl.org/vocab/bio/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?composition
WHERE { ?composition rdf:type ns2:Composition ;
    ns5:date ?date .
FILTER ( ( ?date > 1800 && ?date < 1810 ) ) }
```

Listing 24: *SPARQL query to get compositions within a time range*

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX type: <http://dbtune.org/classical/resource/type/>
PREFIX ns2: <http://purl.org/ontology/mo/>
PREFIX ns5: <http://purl.org/vocab/bio/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?composition
WHERE { ?composition rdf:type ns2:Composition ;
    ns5:place "Salzburg" . }
```

Listing 25: *SPARQL query to get compositions from a given location*

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX void: <http://rdfs.org/ns/void#>
PREFIX DBpedia: <https://www.dbpedia.org/>
PREFIX georss: <http://www.georss.org/georss/>

SELECT DISTINCT ?predicate ?predicateLabel ?value ?valueLabel ?
   coordinates
WHERE {
  DBpedia: void:sparqlEndpoint ?sparqlEndpoint .
  <http://dbtune.org/classical/resource/composer/beethoven_ludwig_van>
      owl:sameAs ?externalResource .
  filter ( regex(str(?externalResource), "dbpedia")) .

  SERVICE ?sparqlEndpoint {
    ?externalResource ?predicate ?value .

    OPTIONAL {
      ?predicate rdfs:label ?predicateLabel .
      FILTER (lang(?predicateLabel) = "en") .
    }
    FILTER (regex(?predicateLabel, "birth place") || regex(?
       predicateLabel, "death place")) .

    OPTIONAL {
      ?value rdfs:label ?valueLabel .
      FILTER (lang(?valueLabel) = "en") .
    }

    FILTER ( IF (isLiteral(?value), lang(?value) = "en", TRUE) ) .

    ?value georss:point ?coordinates .
  }
}
```

Listing 26: *SPARQL query to get a composer's birth and death places and coordinates from DBpedia*

9.4 Void description

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix dcterms: <http://purl.org/dc/terms/> .
@prefix void: <http://rdfs.org/ns/void#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix WSDL_DBTUNE: <https://example.com/wsdl_dbtune/> .
@prefix DBpedia: <https://www.dbpedia.org/> .

## --- our dataset ---

WSDL_DBTUNE:  rdf:type void:Dataset ;
        dcterms:title "WSDL DBTUNE Classical" ;
        void:sparqlEndpoint <https://example.com/wsdl_dbtune/sparql> ;
        void:exampleResource <https://example.com/wsdl_dbtune/composer
           /beethoven_ludwig_van> .

## --- datasets to link to ---

# interlinking to DBpedia:
DBpedia:    rdf:type void:Dataset ;
            foaf:homepage <https://www.dbpedia.org/> ;
            dcterms:title "DBPedia" ;
            void:exampleResource <http://dbpedia.org/resource/
               Ludwig_van_Beethoven> ;
            void:sparqlEndpoint <https://dbpedia.org/sparql/> .

WSDL_DBTUNE:wsdl_dbtune-dbpedia-sameAs  rdf:type void:Linkset ;
                            void:linkPredicate owl:sameAs ;
                            void:target WSDL_DBTUNE: ;
```

```
                void:target DBpedia: .
```

Listing 27: *void.ttl file*