TROMPA: Towards Richer Online Music Public-domain Archives

Deliverable 5.3
TROMPA Processing Library

Grant Agreement nr	770376
Project runtime	May 2018 - April 2021
Document Reference	TR-D5.3-TROMPA Processing Library_v1
Work Package	WP5 - Contributor Environment
Deliverable Type	Report
Dissemination Level	PU- Public
Document due date	30 April 2019
Date of submission	30 April 2019
Leader	UPF
Contact Person	Aggelos Gkiokas (aggelos.gkiokas@upf.edu)
Authors	Aggelos Gkiokas (UPF), Wim Klerkx (VD), Álvaro Sarasúa (VL), David Weigl (MDW), Ingmar Vroomen (CDR), Tim Crawford (GOLD)
Reviewers	Álvaro Sarasúa (VL), Emilia Gomez (UPF)

Executive Summary

Trompa Processing Library (TPL) responsibility is to offer a communication layer between Use Cases, CEapi and WP3 technologies. Each time a use case requests any type of data and metadata, this is done via TPL functionalities which derive from:

- ❖ **The Multimodal Component** which allows a Pilot user to browse (and combine) Contributor Environment (CE) references to public music data, regardless of content type. Moreover the Multimodal Component provides a graphical interface to browse the CE data.
- ❖ **The GraphQL Interface** which provides Pilot developers rich access to the TROMPA dataset, stored as public data references in the CE database.
- ❖ **Extended API Functionalities** which allows the (real-time) definition, creation, completion and consumption of WP3 technology 'jobs' on public music data referenced in the CE database.

The processing workflow for triggering a WP3 algorithm/task is as follows:

- ❖ A WP3 algorithm defines a **task** (algorithm) template in the CE database.
- ❖ The use case can retrieve the potential tasks from CE and can search a certain descriptor for a specific piece of data.
- ❖ If the descriptor of data does not exist, the use case requests the running of a task.
- ❖ On reception of this request, the **Extended API Functionalities** component creates a node for the specific task in the CE.
- ❖ The WP3 process regularly polls the CEapi for new tasks to the CEapi to get notifications on when a new task is requested.
- ❖ On reception of a new task, the WP3 tool retrieves the necessary parameters, validates and starts the task, then updates the task status in the CEapi.
- ❖ The tool updates the task status regularly to reflect progress.
- ❖ Once the WP3 tool is run and processed the requested task, it stores the output in an external repository, then creates a reference node for the storage location in the CEapi and updates the task status to complete.
- ❖ Either by polling or by subscription, the use case receives the 'complete' update.
- ❖ Use case retrieves the result reference and can create additional relations between result and pre-existing nodes for enriching, semantic interlinking and provenance tracking purposes.

The CEapi's main client interface is an implementation of GraphQL. The internal data model of the CE is based on base level types of the schema.org model and a number of schema.org extensions that fit the needs of the TROMPA project. GraphQL broadly supports three types of functionalities that are accessible through the GraphQL API interface:

- ❖ **Queries**: Queries allow to search data (nodes) in the GraphQL database. The queries are used to search and retrieve data nodes in a "read-only" mode.
- ❖ **Mutations**: Contrary to queries, mutations are queries to add, remove or update data in the database.

- ❖ **Subscriptions**: Allow clients to subscribe to changes in the dataset, for example after a particular node was updated or a particular type of node was created. Once such a change event occurs, the client will be notified in real time.

The CE database will contain references to public online musical data in many different formats. The **Multimodal Component** assists Pilot developers to retrieve that assorted data and present it to users in a comprehensive way. To this end, the Multimodal Component offers 'shortcut' access to CEapi GraphQL functionalities plus example User Interface (UI) components that give users access to these functionalities and enable them to consume the rich TROMPA data set. The Multimodal Component offers a GUI to search the TROMPA data set as contained in the CE database, applying the mockups as researched and produced in **Task 5.3 - Multimodal Integration of Music Data**. Currently, the only shortcut functionality that was worked out in the Multimodal Component is a search component that is able to disclose all the public musical data references stored in the CE database.

TROMPA Processing Library also provides a mechanism by which the technologies of WP3 can be invoked by the user pilots (WP6) through the Contributor Environment and the GraphQL API. At its most basic, this integration mechanism offers automation for the following process:

- User Pilot chooses target content, referenced in CE database
- User Pilot user creates a job to run a process on this content
- WP3 Process picks up job
- WP3 Process executes job on target content, creating and storing a result
- WP3 Process writes reference to result in CE database
- User Pilot picks up result
- User Pilot user consumes result

Process jobs are maintained as GraphQL nodes in the database. The generic Component-CE-WP3 interaction solution is based on a schema.org compatible data model that can be broken down into three parts:

- **Template Nodes:** They are maintained by WP3 developers and used by User Pilots. The Template Nodes represent generic algorithmic processes, e.g. "the extraction of Pitch Class Profiles (PCP) features of an audio piece".
- **Instance Nodes**: They are created by User Pilots and maintained by CE. Instance Nodes correspond to specific tasks requested by User Pilots, e.g. "the extraction of PCP features of the audio piece X".
- **Public Nodes**: - They represent the (public) content on which the WP3 process is run (the audio piece X) and the corresponding results (the PCP features).

Each algorithm must "subscribe" it self to the Contributor Environment. The procedure to do so is summarized as:

- ❖ Create a **SoftwareApplication** node in the CE: This node does not correspond to the specific algorithm, but rather to the software/library that hosts the specific algorithm.
- ❖ Create an **EntryPoint** node in the CE: The entry point corresponds to a specific algorithm/method to be run.
- ❖ Create an *actionApplication* relation between **SoftwareApplication** and **EntryPoint**: This actions defines that this EntryPoint is a part of the **SoftwareApplication** node**.**
- ❖ Create a **ControlAction** template node: This **ControlAction** node will be the model for the 'job' created for a specific algorithm process requests. Each request will result in a copy of

this **ControlAction** node to be created (instantiated) which will then represent the 'job' that can be acted on and followed.

- ❖ Create *potentialAction* relation between **EntryPoint** and (template) **ControlAction:** Relates the **ControlAction** template to the specific **EntryPoint**.
- ❖ Create **Property** template node: Corresponds to existing nodes in the CE database, probably referencing a content file at some public repository. These will be the inputs to the WP3 algorithms.
- ❖ Create **PropertyValueSpecification**: Corresponds to a scalar parameter (string, number, on/off checkbox) that needs to be given by the user as 'settings' inputs, in order to tune the algorithm process.
- ❖ Create object relation between **ControlAction** and **Property**/**PropertyValueSpecification** template nodes respectively.

Each algorithm is invoked by the user pilots, using the RequestControlAction.

- ❖ The user pilot makes a RequestControlAction for a specific EntryPoint. The CEapi translates this request by creating a set of nodes on the basis of the **EntryPoint/ControlAction** template and subsequently responds with the thus created **ControlAction**, including its unique *identifier*. With this identifier, the user pilot can then subscribe to the CEapi and and receive a notification each time thus created **ControlAction** is updated.
- ❖ From the WP3 program/algorithm perspective, there are two ways to get notified that a new task has to be run:
    - ➢ Subscribe to the CEapi on RequestControlAction requests on a specific EntryPoint, through the websocket: This functionality offers the possibility to handle user requests for algorithms in real-time.
    - ➢ Frequently check for tasks in the CE: It is possible to query for **ControlAction**s created on the basis of a certain **EntryPoint**. In this way, the software developed under WP3 can check if new tasks have to be run at convenient times or intervals. This functionality offers WP3 software to handle user requests for algorithms in batches.

Since the algorithm has been invoked, it can update the status of **ControlAction** item created during its process until the process is completed:

- ❖ Once the algorithm receives the job, it can update the status of the ControlAction node to 'received'.
- ❖ While the algorithm is running, it can update the status of the ControlAction node in order to provide more information (e.g. 'running') .
- ❖ Once the process has completed, the algorithm process application should write the result to a public location and add a reference to this result in the CE database:
- ❖ With the identifier obtained from the response of the result node (e.g. **DigitalDocument**) creation, create a *result* relation between **ControlAction** and **DigitalDocument**:
- ❖ When the algorithm process application now updates the **ControlAction** actionStatus to 'complete', the process request response cycle is completed.

The process described above provides a flexible framework for communicating WP3 tasks/jobs with the CE data, and the user pilots. In the perspective of WP3 technologies, for running a specific task, two different types of components are involved:

- ❖ **Wrapping software**: This software is responsible for handling the communication and the requests from/to the GraphQL interface of the CE, and trigger the specific **Task software** needed to run.
- ❖ **Task software**: The software that run the actual computations of the task.

In general, these software might be hosted in different servers. This procedure is global (generic) and can be applied to all of the technologies under WP3. In the main body of the deliverable, we will provide more details on the WP3 technologies side, e.g. where these technologies will be hosted and run, and where the results will be saved. In principle, the intention is to run all the **Wrapping Software** components in a dedicated server provided by UPF, which are responsible for handling the communication with the GraphQL interface, the job requests, and trigger and invoking the appropriate programs. However, where the actual computations will be run depends on the developers of the individual tasks. In the rest of the executive summary, we provide some details on the individual tasks and subtasks of WP3.

The technologies under **Task 3.2 - Music Description** (in correspondence with the Deliverable 3.2 - Music Description) involve tools for extracting low / medium level audio descriptors, which will be extracted with the use of Essentia software. Regarding high level descriptors that correspond to Rhythm Descriptors, Music Similarity, Emotion Tag Annotation (sub-sections 2.2.3, 2.2.5 and 2.2.6 of Deliverable - 3.2 Music Description), Symbolic Descriptors (Section 2.3) and Video Descriptors (Section 2.4) and we will follow the same approach with the one described in the previous subsections, with the only difference that the algorithm software will not be Essentia, but inhouse UPF methods or other open source state-of-the-art methods that will be deployed. Regarding singing voice analysis, it will be deployed using different algorithms: Voiceful Cloud's *VoDesc, Essentia* and the TROMPA Choir Singing Analysis algorithm.

Regarding **Task 3.3 - Audio Processing**, The singing synthesis will be integrated using Voctro Labs' Voiceful Cloud service. The motivation for using this platform is twofold: first, it is an existing working solution for singing synthesis that already stores its results in S3 servers, publicly accessible through URLs to which the CE can store references; second, it facilitates the immediate adaptation of the developed technologies in other uses cases for exploitation that may arise beyond the project. This Cloud API service will be extended for TROMPA and will consist of two main tasks:

- ❖ Deploying the **new models** in the cloud servers. As detailed in **Deliverable 3.3 - Audio Processing**, choir singing synthesis has specific requirements beyond those of solo singing synthesis. During the project, new synthesis models will be created from new recordings for multiple languages. These new models will be made accessible through the cloud service.
- ❖ Adapting the API to support the choir case. Currently, the *VoSynth* API accepts monophonic input scores in MusicXML format, as well as in other specific .txt and .json formats for synthesis. For the choir use case, the API will be adapted in order to accept also MEI scores and to support polyphonic material, i.e. receiving a score containing all the voices to be synthesized and outputting one audio file per voice.

For **Task 3.4 - Visual Analysis of Scores**, a machine-readable representation of a musical score can be generated in one of three basic ways: manual encoding, optical music recognition (OMR) or conversion from an existing representation. For several purposes within TROMPA involving the on-screen display of scores we shall be using OMR technology to extract complete or partial machine-readable score representations in the MEI format from scores saved as PDF or other graphical formats. We choose to work with open-source programs which can be modified or adapted to the purposes of TROMPA, which can be run in batch-mode over large collections of music, and which save their results in a form (such as MusicXML) which can be easily converted to MEI for use within TROMPA. For standard modern musical notation we shall be using the open-source program **Audiveris**. Audiveris is generally reliably accurate with good-quality scores in good condition and well photographed/digitised. For the music-scholars' use case within TROMPA, we shall also be working with vocal music from the 16th century. We shall thus use the specialist software **Aruspix**. Similar to other WP3 technologies, each of the software that will be used will be assigned to an **EntryPoint.** The wrapper software (Figure 5.4) will be (possibly) hosted on the UPF server. Regarding the computations components, the infrastructure that will be hosted is to be determined in the future with respect to the computation demands, which depend on the use cases that involve OMR.

Task 3.5 - Alignment of Musical Resources concerns the interlinking of multimodal representations of a musical expression at different levels of granularity. Various algorithms and software packages to accomplish this task are described in **Deliverable 3.5 - Multimodal Music information Alignment**. For present development, we are focussing our efforts on the MAPS tool ("Matcher for Alignment of Performance and Score") under ongoing in-house development at MDW, which has the advantage of natively supporting MEI. MAPS will be called through a wrapper layer ("MAPS wrapper") responsible for accepting new jobs (e.g., performances) from TROMPA users, registering these within the CE, and spawning MAPS instances to complete the jobs. MAPS instances are processor intensive and should thus be deployed on a dedicated high-performance server - potentially at UPF. The MAPS wrapper is a comparatively light-weight web service that could be deployed separately, e.g. at MDW.

Task 3.6 - Multimodal Cross Linking comprises of two different pieces of software, the user interface (Authority registration and linking) and the web scraper:

❖ **Authority registration and linking**: This software has a user interface to create and describe web-resources and the relations between entities and resource types. For example Wikidata will be registered as authority that provides Person Entities, Musical Work Entities and other entities that will be stored in the CE.

❖ **Web scraper**: The web scraper will retrieve entity information from the in the CE stored web-resource and finds links to other registered authorities. It will save the referenced entities in the CE.

The user interface (Authority registration and linking) interacts with contributors and the CE. On the other hand, the web scraper interacts between the CE and the web resources of registered authorities. The multimodal cross linking will have use cases where the linked data can be used as background reference or where data can be enriched for the end user. This is primarily with cases for music scholars and music enthusiasts. Both pieces of software will run as micro services in a cloud environment under subscription of Trompa. The authority registration and linking software will register user input from a web-interface and store the data in the CE. The web scraper takes data from web-resources of registered authorities and saves the linked data in the CE.

Rather than being a concrete piece of software, TROMPA Processing Library is a collection of Contributor Environment functionalities that allow the interaction of WP3 components with the CE data, the different software components that correspond to the various WP3 tasks, and the organization amongst them. For the next period of the project, we plan to develop the TPL as follows:

❖ **Preliminary Development** (M13 - 18): By the end of this period we will test all of the individual components of WP3 and validate the correct communication with the CE: assignment of tasks from the CE, execution of tasks, storage of data.

- **First Working Version** (M19-M22): By the end of this period we will provide a first full version of the TPL. This will be delivered 2 months prior to M24 and MS3, where the first version of the working prototypes of the pilots will be delivered in order to facilitate the development of the pilots.
- **Incorporation of Latest WP3 Components** (M23-24): TPL will incorporate the final versions of for Task 3.3 - Audio Processing and Task 3.5 - T3.5 Alignment of musical resources.
- **Incorporation of Final WP3 Components** (M25-M30): TPL will incorporate the final versions of for Task 3.4 - Visual Analysis of Scanned Scores and Task 3.6 - Multimodal Cross Linking
- **Final Version** (M31-M34): Final adaptations/debugging. There will be an effort to have most of the components centralised for ensuring sustainability after the end of the project.

Version Log		
#	Date	Description
v0.1	29 March 2019	Initial draft from UPF
v0.2	4 April 2019	Integrated initial contributions from VD
V0.3	8 April 2019	Added Section 5
v0.3.1	9 april 2019	Added Section 4
v0.4	26 April 2019	Completed Sections 4 & 5
v0.5	26 April 2019	Added Section 6
v0.6	27 April 2019	Minor corrections to the text/formatting
v0.7	29 April 2019	Added Executive Summary
v0.8	29 April 2019	References fixed
v1.0	30 April 2019	Final version submitted to EU

Table of Contents

1. Introduction

This deliverable provides functionalities for embeddable descriptions and synthesis of music data coming from supported music data repositories.

In the project proposal, Task 5.3 responsibilities are described as Multimodal Integration and as Processing Library (D5.3). Together, these responsibilities comprise common access to (combinations of) public music data contained in the Contributor Environment (CE) database regardless of content-type and common access to WP3 algorithm processes to be run against (combinations of) this public music data.

With the setup of the CE as described in D5.1 - Data Infrastructure[1], these functionalities are not designed to be all encapsulated within a single software library, but they are to be delivered by the integration of functionalities of both T5.1 and T5.3:

❖ GraphQL implementation features of the Contributor Environment API (CEapi).
❖ The Multimodal Component, offering ‘shortcut’ access to CEapi GraphQL functionalities, like a search interface. The Multimodal Component also provides a graphical interface to browse the CE data.
❖ An extension of CE-api functionalities, offering a generic integration solution for WP3-CEApi-Pilot interaction.
❖ Example code for WP3-CEApi-Pilot integration, currently consisting of a working example of Verovio MusicXML->MEI conversion on eligible data contained in the CE database (Annex B).

This deliverables organizes as follows. First, we present an overview on the TROMPA Processing Library and a short description of its components (Section 2). In Sections 3, 4 and 5 respectively we provide details of the above-mentioned functionalities. Section 6 concludes this deliverable with the planning of the TROMPA Processing Library for the next period.

2. The TROMPA Processing Library Overview

Trompa Processing Library (TPL) responsibility is to offer a communication layer between Use Cases, CEapi and WP3 technologies. An overview of the TROMPA Processing Library workflow is shown in Figure 2.1. Each time a use case requests any type of data and metadata, this is done via TPL functionalities which derive from:

❖ **The Multimodal Component,** which allows a Pilot user to browse (and combine) CE references to public music data, regardless of content type. Moreover the Multimodal Component provides a graphical interface to browse the CE data.
❖ **The GraphQL Interface,** which provides Pilot developers rich access to the TROMPA dataset, stored as public data references in the CE database.
❖ **Extended API Functionalities,** which allows the (real-time) definition, creation, completion and consumption of WP3 technology ‘jobs’ on public music data referenced in the CE database.

[1]https://trompamusic.eu/deliverables/TR-D5.1-Data_Infrastructure_v1.pdf

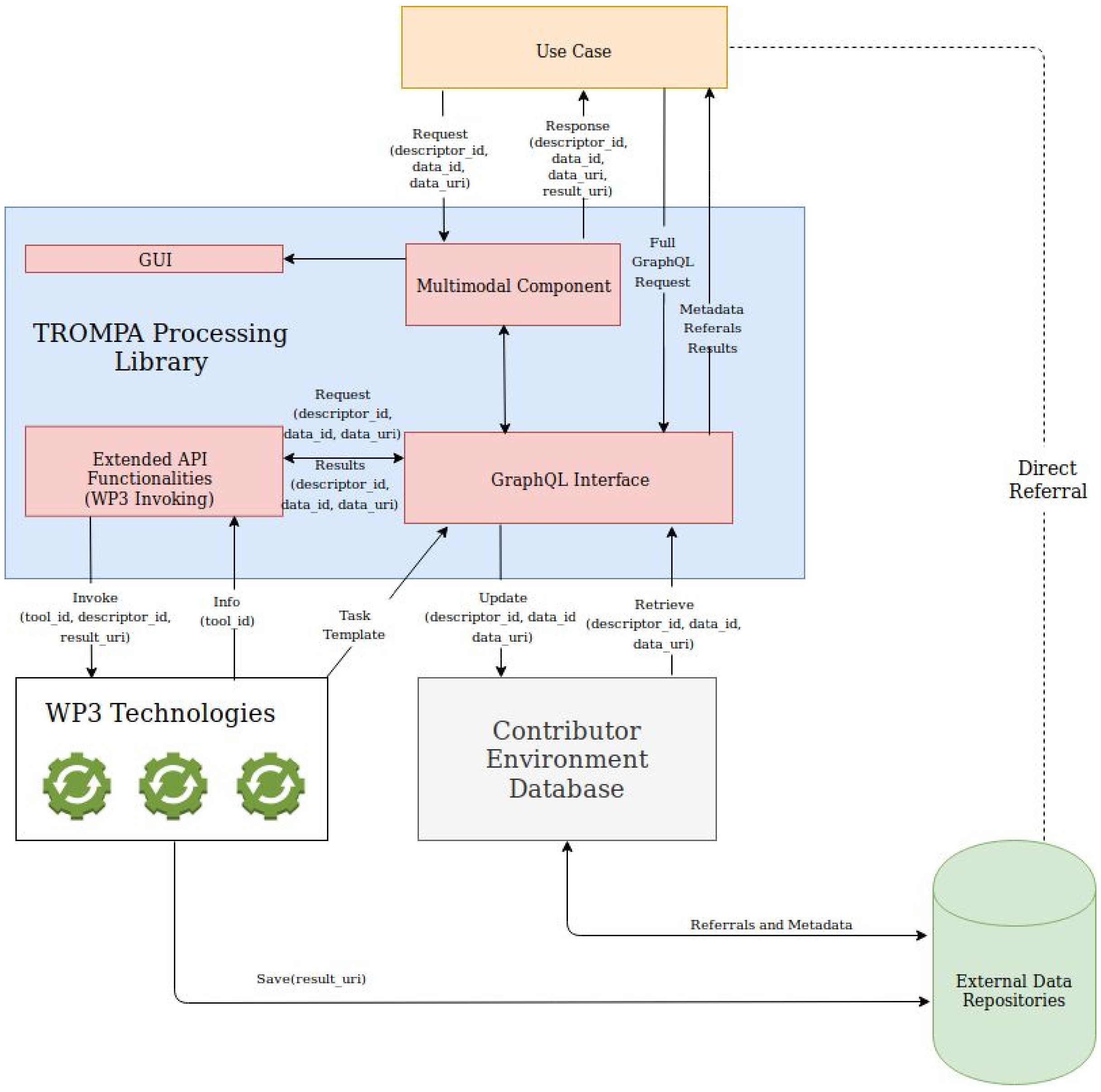

Use Case
Request (descriptor_id, data_id, data_uri)
Response (descriptor_id, data_id, data_uri, result_uri)
Full GraphQL Request
Metadata Referals Results
Direct Referral
GUI
Multimodal Component
TROMPA Processing Library
Request (descriptor_id, data_id, data_uri)
Results (descriptor_id, data_id, data_uri)
Extended API Functionalities (WP3 Invoking)
GraphQL Interface
Invoke (tool_id, descriptor_id, result_uri)
Info (tool_id)
Task Template
Update (descriptor_id, data_id data_uri)
Retrieve (descriptor_id, data_id, data_uri)
WP3 Technologies
Contributor Environment Database
Referrals and Metadata
Save(result_uri)
External Data Repositories

Figure 2.1 TROMPA Processing Library Workflow

The processing workflow for triggering a WP3 algorithm/task is as follows:

- ❖ Initially, the WP3 algorithm maintainer defines a 'job' template in the CE database, using extended CEapi functionality for this purpose. A job template corresponds to a specific **task** (algorithm) to be run.
- ❖ The use case can retrieve the potential tasks to be run on a specific type of data from CEapi and during its execution can search a certain descriptor for a specific piece of data. This can be done by either using the TROMPA Multimodal Component or by executing a Full GraphQL Query.
- ❖ If the task was already run on this piece of data, the use case can retrieve and show the result immediately. If the piece of data does not exist, the use case requests the running of a 'task' on the selected piece of data by submitting the appropriate GraphQL request as defined by the WP3 algorithm maintainer.

- On reception of this request, the **Extended API Functionalities** component creates the necessary 'job' nodes as defined by the WP3 algorithm maintainer.
- Either the WP3 process (tool) regularly polls the CEapi for new 'jobs' created, or subscribes to 'job' creation triggers via a websocket. In principle, these tools are **not** inside the CE facilities.
- On reception of a new job, the WP3 tool retrieves the necessary parameters, validates and starts the task, then updates the task status in the CEapi. To track the progress of the execution, either the use case regularly polls the CEapi for task updates, or subscribes to update triggers via a websocket.
- Optionally, the tool updates the task status regularly to reflect progress, and optionally, the use case polls or subscribes to task updates to remain informed of progress updates.
- Once the WP3 tool is run and processed the requested task, it stores the output in an external repository, then creates a reference node for the storage location in the CEapi, then creates a *result* relation between task node and reference node.
- Once the result is related to the task node, the WP3 tool updates the task status to complete.
- Either by polling or by subscription, the use case receives the 'complete' update.
- Use case retrieves the result reference and can create additional relations between result and pre-existing nodes for enriching, semantic interlinking and provenance tracking purposes.

The proposed workflow provides the flexibility of running algorithms 'on demand', i.e. running a specific algorithm on a specific data item when requested by the use cases, as well as in a 'batch' mode, i.e. running an algorithm in a large amount of data prior to the execution of a use case.

3. GraphQL Interface

3.1 Overview

The CEapi's main client interface is an implementation of GraphQL. GraphQL[2] is an open standard API query language that is designed to allow clients flexible API access to datasets but also to processes, responding either with customized data objects aggregated from data from the database or from secondary data stores or processes. Its online specification manual[3] can provide detailed background information for the following sections.

The internal data model of the CE is based on base level types of the schema.org[4] model and a number of schema.org extensions that fit the needs of the TROMPA project. The default schema.org properties were supplemented with properties derived from well known ontologies. These supplementary properties express CE needs for metadata, interlinking, internationalization, and provenance tracking.

Each "node" in the internal data model of the CE corresponds to a certain item of the ontology (e.g. Artist/ Creative Work), which is illustrated in Figure 3.1. A detailed description of the ontology is can be found in the document *CE Data Import Guidelines*. This document will be made available

[2] https://graphql.org/
[3] https://graphql.org/learn/
[4] https://schema.org/

online as the development of the Contributor Environment progresses. GraphQL broadly supports three types of functionalities that are accessible through the GraphQL API interface:

- **Queries**: Queries allow to search data (nodes) in the GraphQL database. The queries are used to search and retrieve data nodes in a "read-only" mode.
- **Mutations**: Contrary to queries, mutations are queries to add, remove or update data in the database.
- **Subscriptions**: Allow clients to subscribe to changes in the dataset, for example after a particular node was updated or a particular type of node was created. Once such a change event occurs, the client will be notified in real time.

In the rest of this section we will provide a brief but comprehensive description of three types of functionalities. Complete examples of query and mutation functionalities are provided in Annex A.

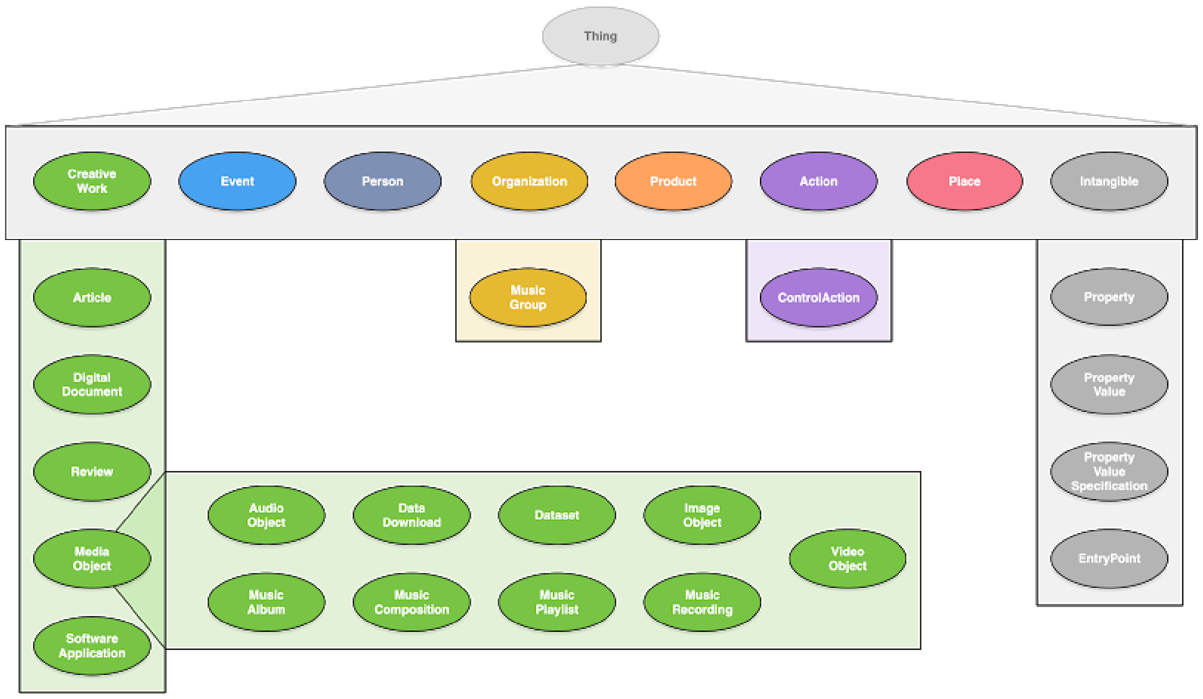


Figure 3.1 - CE Data Model

3.2 Queries

We define two main types of queries, the **simple queries** and complex **queries**. Typically, a simple query consists of:

- The name of the query (optional).
- Type of entity for which is queried (i.e. one of the entities of the taxonomy provided in Figure 4.1, e.g. a **Music Recording**).
- Conditions: the conditions under which this entity is queried. For example, "a music recording (entity) of **Composer** Gustav Mahler (condition)". Conditions are optional.
- List of properties to be included in the response. For example, "get the **Title** and **Description** (response types) of the **Music Recordings** (entity) of **Composer** Gustav Mahler (condition).

The result typically consists of a json object containing:

- The "data" object with the result(s).
- The name of the query responded to.

- The actual data, corresponding to the list of properties to be included.

Regarding complex queries, the difference to simple queries is that the properties requested by the query can be complex, i.e. composition from other nodes. Examples of queries can be found in Annex A-1.

3.3 Mutations

Mutations are queries that add, update or remove data in the database. There are seven types of GraphQL mutations described below:

- **Node creation**: A create mutation typically consists of
    - a list of scalar parameters that correspond to the type properties for which the value needs to be set and
    - a list of properties to be returned once the node is created.
- **Node update:** The update query is similar to the node creation, with the difference that the identifier of the node to be updated must be passed along with the update parameters. Properties that are left out will not be updated.
- **Node deletion**: When deleting a node, only the identifier of the node can be passed as a parameter. When a node gets deleted, all its incoming and outgoing relations to other nodes will also be deleted.
- **Add relation between nodes**: For each relation, which is a property containing another node, there is a dedicated mutation query. For example there is a predefined mutation for relating an event node to a composer named **AddEventComposer**:

```
AddEventComposer(
     from: _EventInput
     to: _LegalPersonInput
)
```

This mutation takes as input an Event node and a **LegalPerson** node and adds its relation to the database. Moreover, it is rational and not possible to create multiple relations of the same type between the same nodes.

3.4 Subscriptions

Currently, subscriptions can be created for mutations on **ControlAction** type nodes and for **ControlAction** nodes created through the RequestControlAction request. In the final version of this deliverable, we intend to allow subscriptions to be created for mutations on nodes of any type.

4. Multimodal Component

4.1 Overview

The CE database will contain references to public online musical data in many different formats. The Multimodal Component assists Pilot developers to retrieve that assorted data and present it to users in a comprehensive way.

To this end, the Multimodal Component offers ‘shortcut’ access to CEapi GraphQL functionalities plus example UI components that give users access to these functionalities and enable them to consume the rich TROMPA data set.

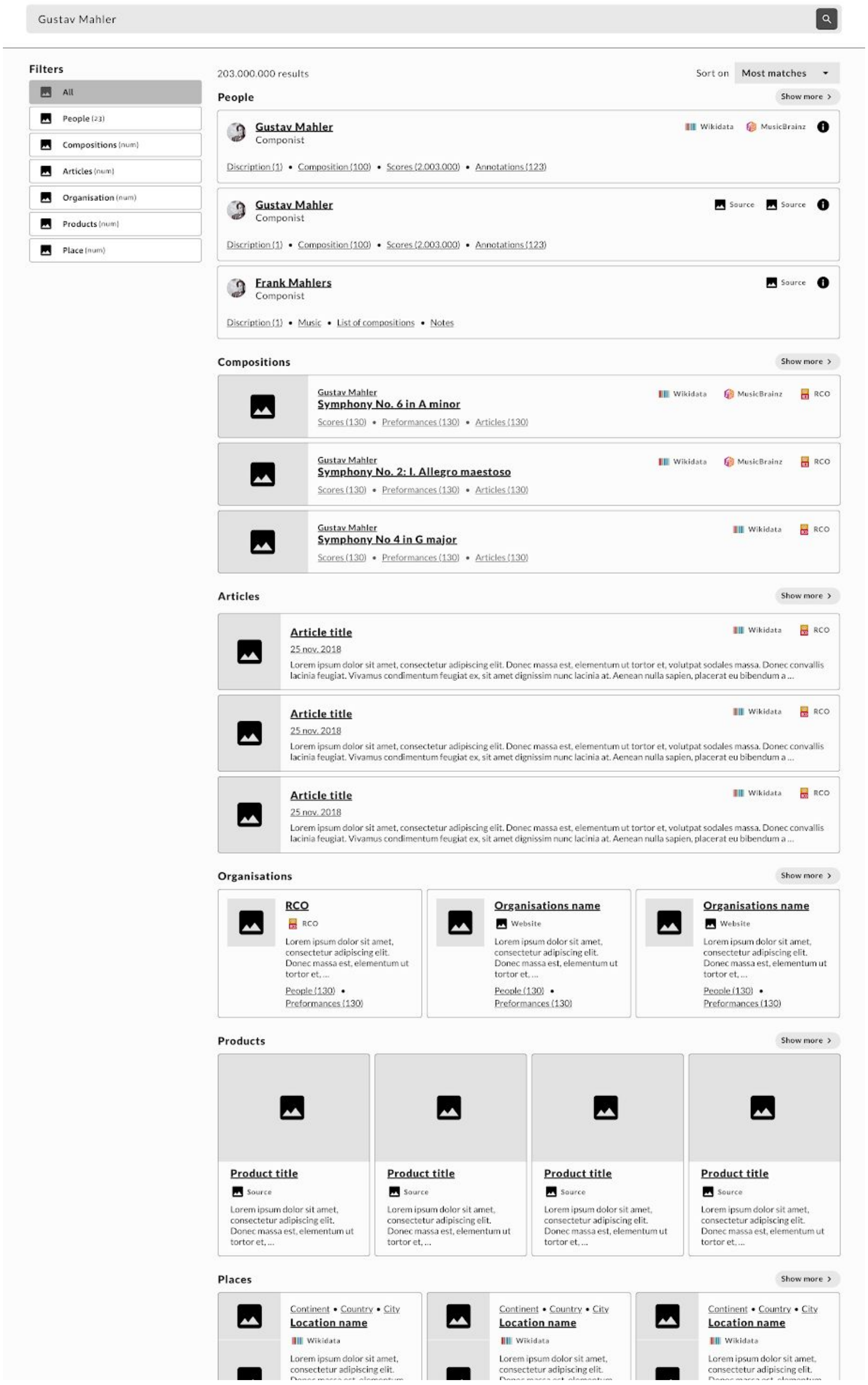
4.2 Communication with the Use Cases

Currently, the Multimodal Component consists of a React application that implements search functionality on CE database content through the GraphQL interface. A Use Case developer can include this React component and use (parts of) its code and UI elements where they fit Use Case specific functionalities. The React component can also serve as a working example for developers to build Use Case specific implementations on the same GraphQL functionalities or to use another technology stack.

4.3 A Graphical Interface for Accessing Data

The Multimodal Component offers a GUI to search the TROMPA dataset as contained in the CE database, applying the mockups as researched and produced in **Task 5.3 - Multimodal Integration of Music Data**. Figure 4.1 presents the output of the query of ‘Gustav Mahler’ to the CE database. On the left side we can see the different categories of the items matching the query. Note that these categories are the same categories of the CE data model which is illustrated in Figure 3.1. For each result of the query additional information is provided, as for example the source of item (e.g. wikidata, musicbrainz), and links to related items. For instance a composer is linked to her/his compositions, and the compositions are linked to scores and performances. The end user can get into more deep in the structure of the data, as for example by filtering only by People entity, as shown in Figure 4.2. Figure 4.3 shows a screenshot of a demo that is available online[5]. The demo site will be updated regularly to reflect the latest developments of the Multimodal Component search functionalities.

[5] https://trompa-ce-search.netlify.com/



Gustav Mahler

Filters

All
People (23)
Compositions (num)
Articles (num)
Organisation (num)
Products (num)
Place (num)

203.000.000 results

Sort on Most matches

People

Show more

Gustav Mahler
Componist
Wikidata MusicBrainz
Discription (1) • Composition (100) • Scores (2.003.000) • Annotations (123)

Gustav Mahler
Componist
Source Source
Discription (1) • Composition (100) • Scores (2.003.000) • Annotations (123)

Frank Mahlers
Componist
Source
Discription (1) • Music • List of compositions • Notes

Compositions

Show more

Gustav Mahler
Symphony No. 6 in A minor
Wikidata MusicBrainz RCO
Scores (130) • Preformances (130) • Articles (130)

Gustav Mahler
Symphony No. 2: I. Allegro maestoso
Wikidata MusicBrainz RCO
Scores (130) • Preformances (130) • Articles (130)

Gustav Mahler
Symphony No 4 in G major
Wikidata RCO
Scores (130) • Preformances (130) • Articles (130)

Articles

Show more

Article title
Wikidata RCO
25 nov. 2018
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, volutpat sodales massa. Donec convallis lacinia feugiat. Vivamus condimentum feugiat ex, sit amet dignissim nunc lacinia at. Aenean nulla sapien, placerat eu bibendum a ...

Article title
Wikidata RCO
25 nov. 2018
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, volutpat sodales massa. Donec convallis lacinia feugiat. Vivamus condimentum feugiat ex, sit amet dignissim nunc lacinia at. Aenean nulla sapien, placerat eu bibendum a ...

Article title
Wikidata RCO
25 nov. 2018
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, volutpat sodales massa. Donec convallis lacinia feugiat. Vivamus condimentum feugiat ex, sit amet dignissim nunc lacinia at. Aenean nulla sapien, placerat eu bibendum a ...

Organisations

Show more

RCO
RCO
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, ...
People (130) •
Preformances (130)

Organisations name
Website
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, ...
People (130) •
Preformances (130)

Organisations name
Website
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, ...
People (130) •
Preformances (130)

Products

Show more

Product title
Source
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, ...

Product title
Source
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, ...

Product title
Source
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, ...

Product title
Source
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec massa est, elementum ut tortor et, ...

Places

Show more

Continent • Country • City
Location name
Wikidata
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Continent • Country • City
Location name
Wikidata
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Continent • Country • City
Location name
Wikidata
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Figure 4.1 Search mockup - initial search

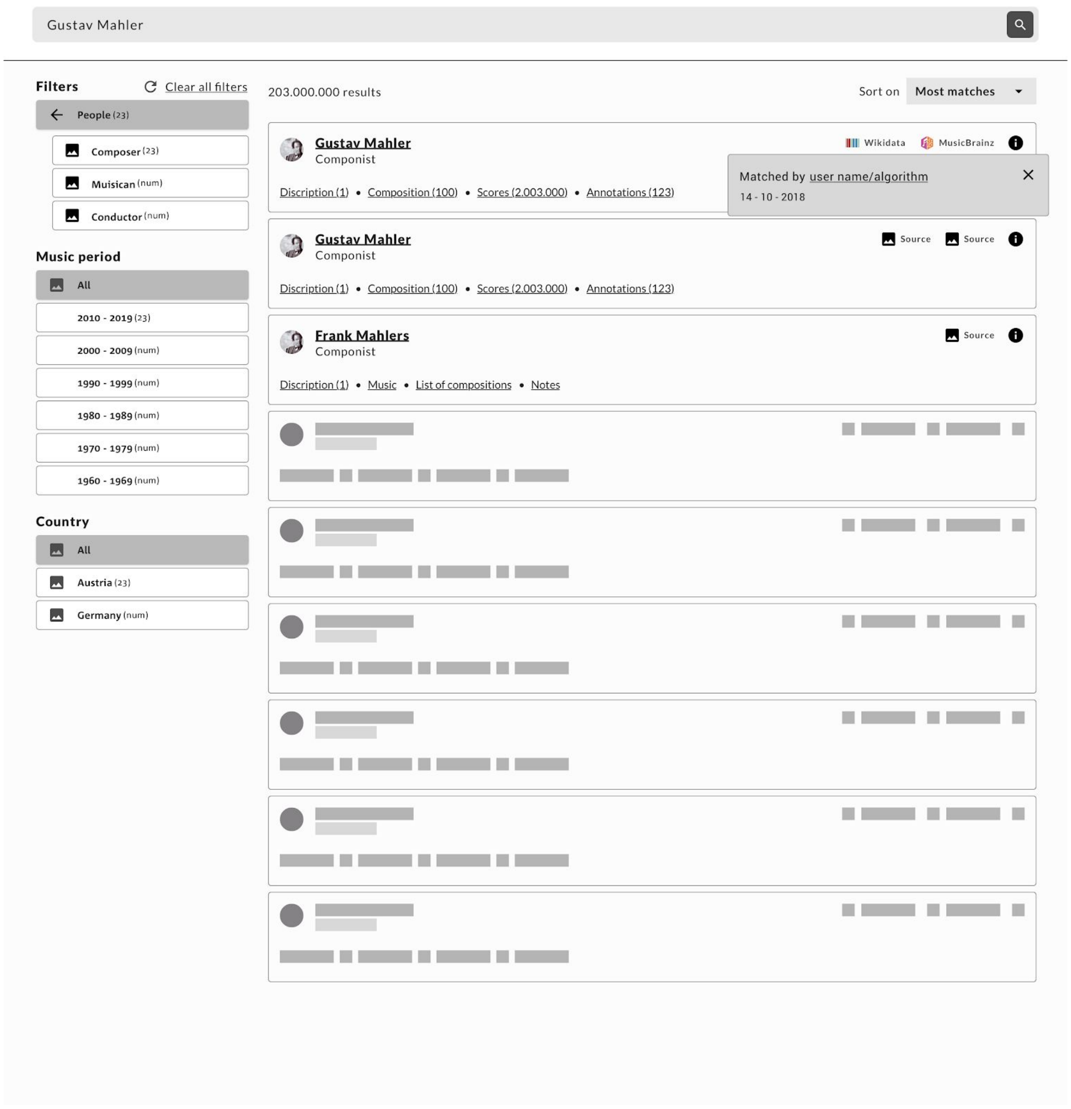
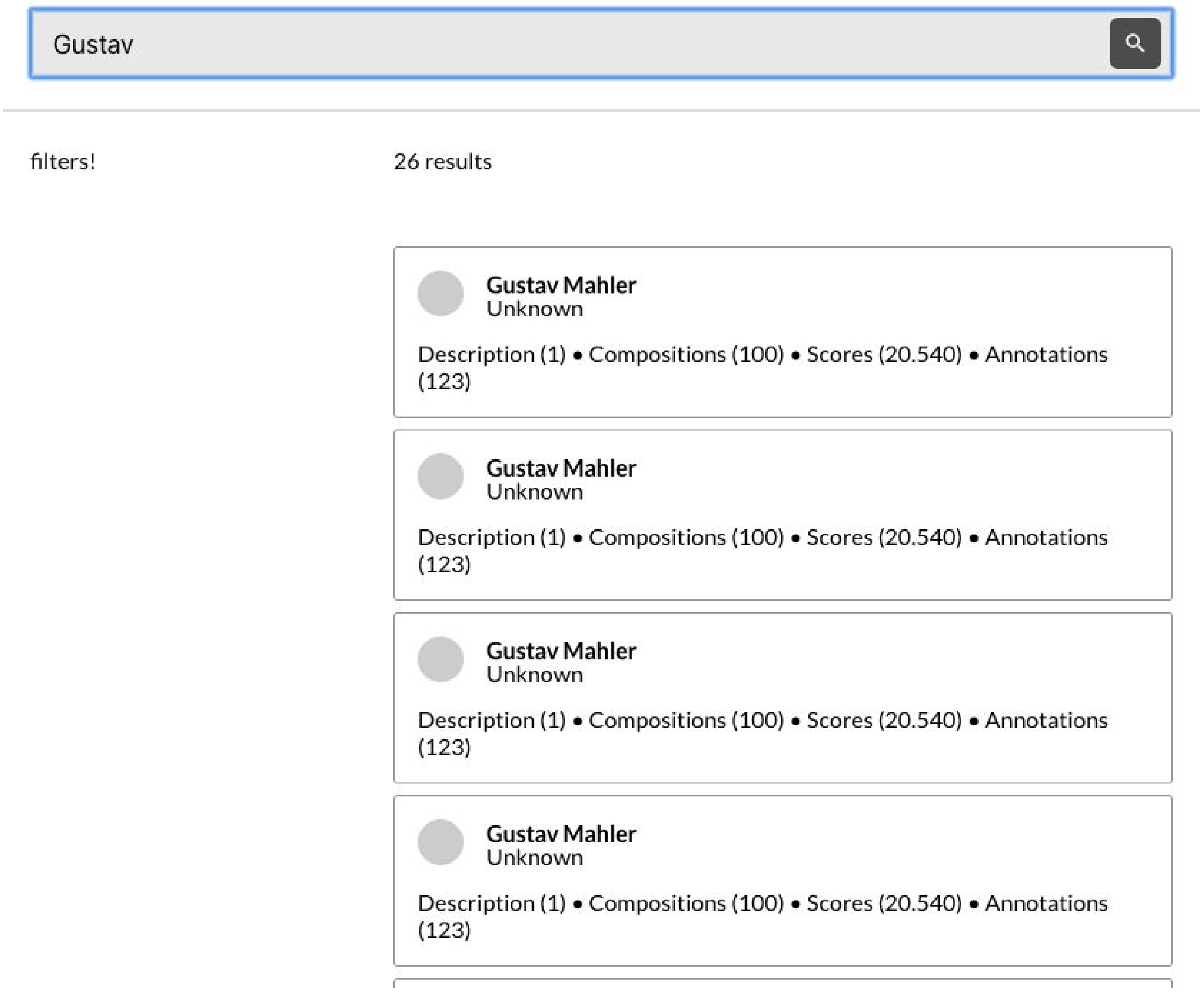


Figure 4.2 Search mockup - refined search on People



Gustav

filters!

26 results

Gustav Mahler
Unknown

Description (1) • Compositions (100) • Scores (20.540) • Annotations (123)

Gustav Mahler
Unknown

Description (1) • Compositions (100) • Scores (20.540) • Annotations (123)

Gustav Mahler
Unknown

Description (1) • Compositions (100) • Scores (20.540) • Annotations (123)

Gustav Mahler
Unknown

Description (1) • Compositions (100) • Scores (20.540) • Annotations (123)

Figure 4.3 Search implementation - unrefined search on People

4.4 Shortcut functionality to GraphQL

Currently, the only shortcut functionality that was worked out in the Multimodal Component is a search component that is able to disclose all the public musical data references stored in the CE database. At this point, there are no actionable plans for other CEapi functionalities to be streamlined for generic application in Use Case development. As the GraphQL interface is very versatile and flexible, it allows rich access to the CE database. For most applications, the GraphQL interface will provide sufficient functionality to probe the CE database.

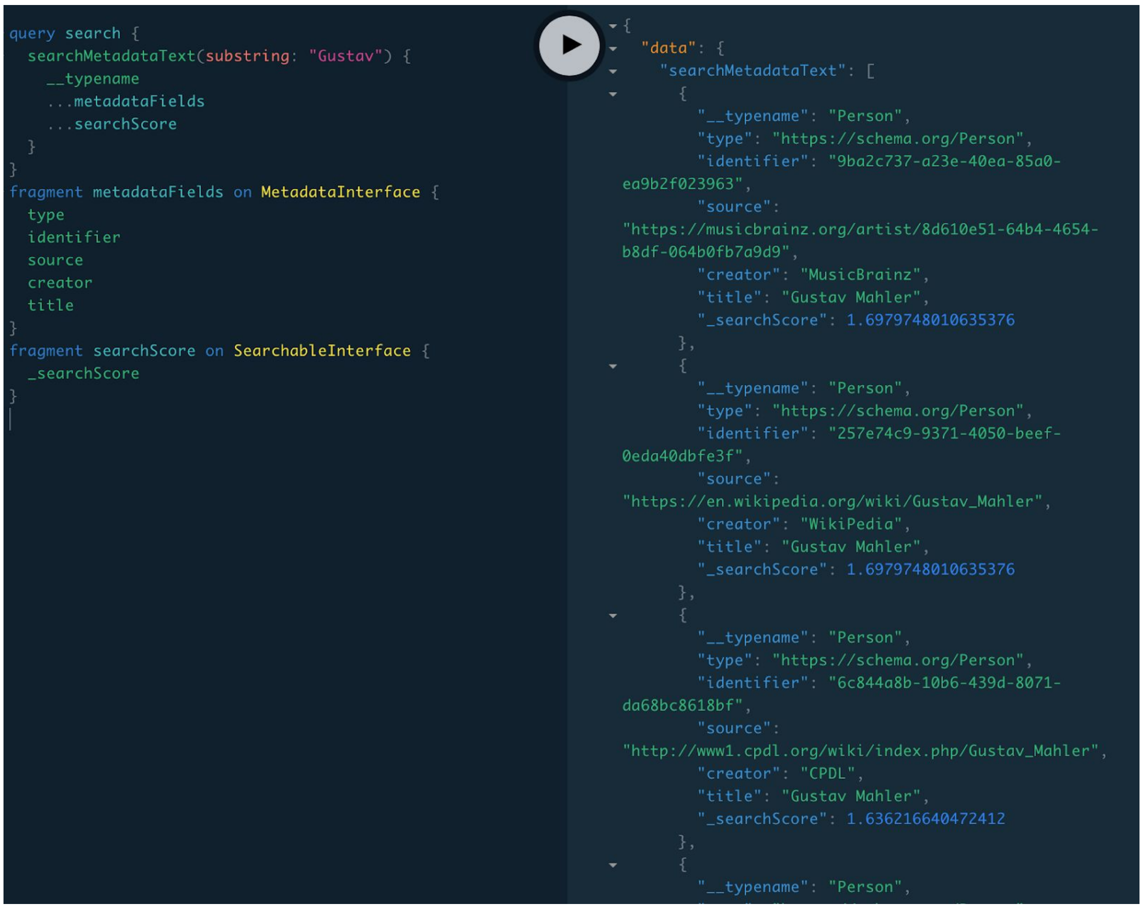
Once WP3 tasks are being integrated with WP5 and start making interlinked contributions to the TROMPA dataset, it is possible that additional generic functionalities can be conceived of and added to the CEapi. Adding matching functionality to the Multimodal Component should help Pilot developers to integrate these functionalities in WP6 tasks with options that cannot be provided through the GraphQL interface directly.

The search interface described in 4.3 consists of a GUI and React code that makes use of CEapi functionality through the GraphQL interface. Here follows an overview of this CEapi search functionality, as exploited by the Multimodal Component. Of course, this functionality can be exploited by use case developers in other ways than by implementing parts of the Multimodal Component code.

4.4.1 Basic metadata search

The searchMetadataText query generates results on the basis of a fulltext index[6] on the *title, description, subject* and *creator* properties of all CE schema types that extend the **MetadataInterface**. The results will be ordered according to search scores generated along with the database query results, as shown in Figure 4.4.

[6] https://neo4j.com/docs/cypher-manual/current/schema/index/#schema-index-fulltext-search



```
query search {
  searchMetadataText(substring: "Gustav") {
    __typename
    ...metadataFields
    ...searchScore
  }
}
fragment metadataFields on MetadataInterface {
  type
  identifier
  source
  creator
  title
}
fragment searchScore on SearchableInterface {
  _searchScore
}
```

```
{
  "data": {
    "searchMetadataText": [
      {
        "__typename": "Person",
        "type": "https://schema.org/Person",
        "identifier": "9ba2c737-a23e-40ea-85a0-
ea9b2f023963",
        "source":
"https://musicbrainz.org/artist/8d610e51-64b4-4654-
b8df-064b0fb7a9d9",
        "creator": "MusicBrainz",
        "title": "Gustav Mahler",
        "_searchScore": 1.6979748010635376
      },
      {
        "__typename": "Person",
        "type": "https://schema.org/Person",
        "identifier": "257e74c9-9371-4050-beef-
0eda40dbfe3f",
        "source":
"https://en.wikipedia.org/wiki/Gustav_Mahler",
        "creator": "WikiPedia",
        "title": "Gustav Mahler",
        "_searchScore": 1.6979748010635376
      },
      {
        "__typename": "Person",
        "type": "https://schema.org/Person",
        "identifier": "6c844a8b-10b6-439d-8071-
da68bc8618bf",
        "source":
"http://www1.cpdl.org/wiki/index.php/Gustav_Mahler",
        "creator": "CPDL",
        "title": "Gustav Mahler",
        "_searchScore": 1.636216640472412
      },
      {
        "__typename": "Person",
```

Figure 4.4 CEapi Search implementation - basic GraphQL search query

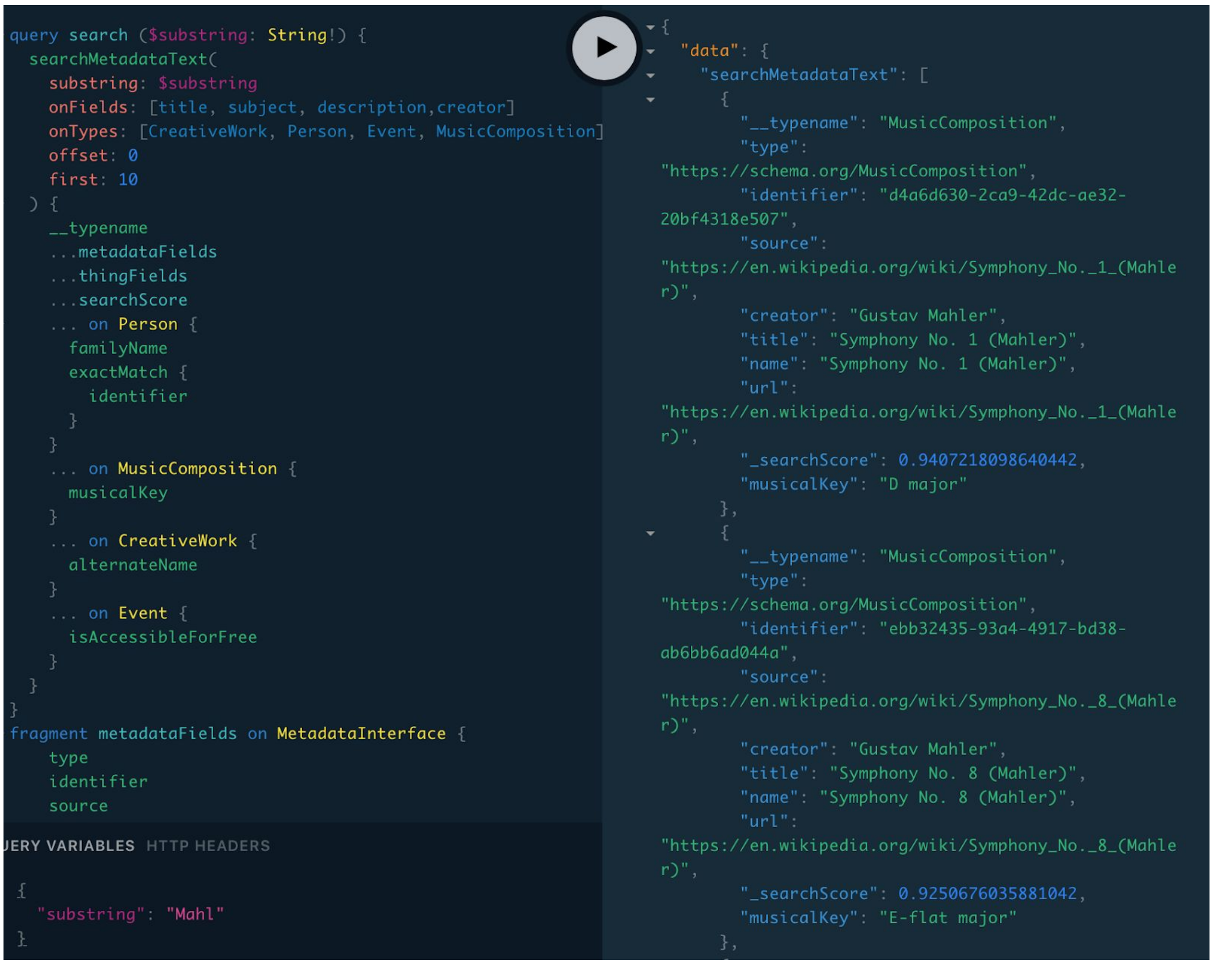
4.4.4 Advanced metadata search

The fulltext index is maintained on a selection of properties, which in turn is maintained on a selection of schema types (extending MetadataInterface). This setup allows the search functionality to be enhanced by optionally filtering on these properties and schema types:

- ❖ Allow to search on a subset of indexed properties
- ❖ Allow to search on a subset of indexed schema types
- ❖ Allow to search on any combination of those two subsets

As the CE data model is based on a semantically sound schema aimed at providing structured data on the internet[7], this quite naturally leads to the ability to filter the search results in a semantic way (Figure 4.5)

[7] https://en.wikipedia.org/wiki/Schema.org



```
query search ($substring: String!) {
  searchMetadataText(
    substring: $substring
    onFields: [title, subject, description,creator]
    onTypes: [CreativeWork, Person, Event, MusicComposition]
    offset: 0
    first: 10
  ) {
    __typename
    ...metadataFields
    ...thingFields
    ...searchScore
    ... on Person {
      familyName
      exactMatch {
        identifier
      }
    }
    ... on MusicComposition {
      musicalKey
    }
    ... on CreativeWork {
      alternateName
    }
    ... on Event {
      isAccessibleForFree
    }
  }
}
fragment metadataFields on MetadataInterface {
    type
    identifier
    source
```

ERY VARIABLES HTTP HEADERS

```
{
  "substring": "Mahl"
}
```

```
{
  "data": {
    "searchMetadataText": [
      {
        "__typename": "MusicComposition",
        "type":
"https://schema.org/MusicComposition",
        "identifier": "d4a6d630-2ca9-42dc-ae32-
20bf4318e507",
        "source":
"https://en.wikipedia.org/wiki/Symphony_No._1_(Mahle
r)",
        "creator": "Gustav Mahler",
        "title": "Symphony No. 1 (Mahler)",
        "name": "Symphony No. 1 (Mahler)",
        "url":
"https://en.wikipedia.org/wiki/Symphony_No._1_(Mahle
r)",
        "_searchScore": 0.9407218098640442,
        "musicalKey": "D major"
      },
      {
        "__typename": "MusicComposition",
        "type":
"https://schema.org/MusicComposition",
        "identifier": "ebb32435-93a4-4917-bd38-
ab6bb6ad044a",
        "source":
"https://en.wikipedia.org/wiki/Symphony_No._8_(Mahle
r)",
        "creator": "Gustav Mahler",
        "title": "Symphony No. 8 (Mahler)",
        "name": "Symphony No. 8 (Mahler)",
        "url":
"https://en.wikipedia.org/wiki/Symphony_No._8_(Mahle
r)",
        "_searchScore": 0.9250676035881042,
        "musicalKey": "E-flat major"
      },
```

Figure 4.5 CEapi Search implementation - filtered GraphQL search query

This way, the filtering options allow a user to search for **Person**s with a certain name, or on **MusicComposition**s answering to certain description keys, etcetera.

The searchMetadataText functionality also allows Use Case developers to present their users with well defined subsets of TROMPA data that could fit a specific use case, like only **DigitalDocument**s or only **AudioObjects** and **VideoObject**s created by a certain Person.

5. Extended API Functionalities to support WP3-WP6 integration

This section provides details on the mechanism by which the technologies of WP3 can be invoked by the user pilots (WP6) through the Contributor Environment and the GraphQL API. At its most basic, this integration mechanism offers automation for the following process:

- User Pilot chooses target content, referenced in CE database
- User Pilot user creates a job to run a process on this content
- WP3 Process picks up job
- WP3 Process executes job on target content, creating and storing a result
- WP3 Process writes reference to result in CE database
- User Pilot picks up result
- User Pilot user consumes result

In the following subsections we will describe the technical details of this mechanism.

5.1 Data Model for WP3 Processes

5.1.1 Overview

Process jobs are maintained as GraphQL nodes in the database. A subscription[8] mechanism can enable WP3 processes actively updated on job creation and status updates in real time. This way, the CE becomes the intermediary of Use Case-WP3 interactions. This offers a standardized solution for integration and ensures WP3 produced data is referenced and gets interlinked with the larger TROMPA dataset. The generic Component-CE-WP3 interaction solution is based on a schema.org[9] compatible data model that can be broken down into three parts:

- **Template Nodes:** They are maintained by WP3 developers and used by User Pilots. The Template Nodes represent generic algorithmic processes, e.g. “the extraction of Pitch Class Profiles (PCP) features of an audio piece”.
- **Instance Nodes**: They are created by User Pilots and maintained by CE. Instance Nodes correspond to specific tasks requested by User Pilots, e.g. “the extraction of PCP features of the audio piece X”.
- **Public Nodes**: - They represent the (public) content on which the WP3 process is run (the audio piece X) and the corresponding results (the PCP features).

[8] https://graphql.org/blog/subscriptions-in-graphql-and-relay/
[9] https://schema.org/

5.1.2 Template nodes

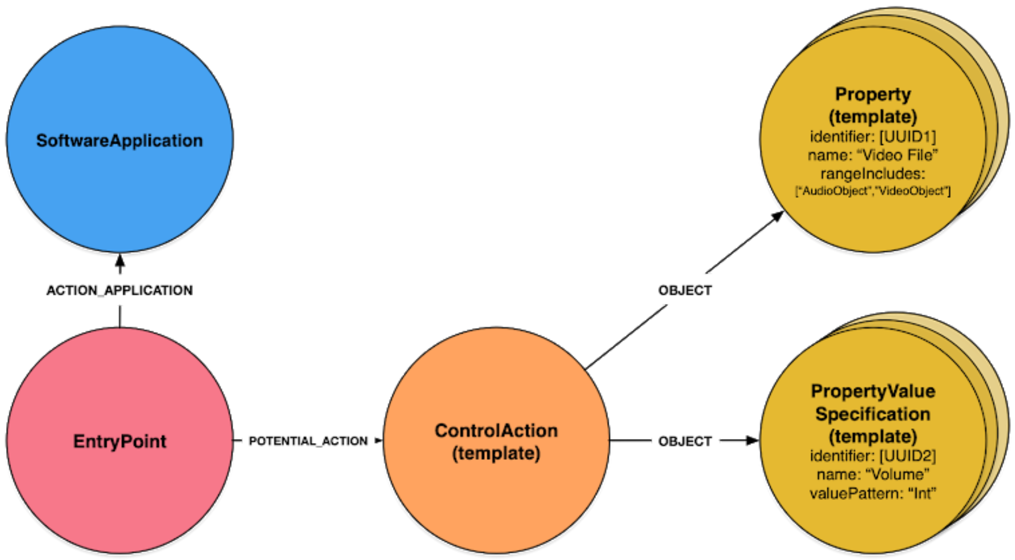


Figure 5.1. Template nodes schema.

Figure 5.1 illustrates the template notes and how they relate to an "algorithmic process". Each WP3 process application, e.g. an alignment tool, will need a **SoftwareApplication**[10] node in the database. This is a node that can tie a number of available algorithmic processes to one of the TROMPA participants or to a software bundle.

Each of the available algorithmic processes needs to be represented as an **EntryPoint**[11] node. This entry point enables the user pilot to request, monitor and control the running of a process. An entry point is what is presented by the Component as an available functionality, like the automatic analysis of a recording or the annotation of a digital score. The **EntryPoint** needs to be related to the **SoftwareApplication** through the *actionApplication*[12] property. The **ControlAction**[13] is the `template` for a user's request for a certain process to be run and is related to the **EntryPoint** through the *potentialAction* property. It is like a super-class for a potential job that needs to be carried out by the process represented by the **EntryPoint**. For a process job to be able to run, probably a number of parameters need to be passed along to tell the process on what target data to act on, plus some parameters for tuning the process or naming of the results. Any number of required or non-required scalar arguments (numbers, strings etc.) can be set up by adding **PropertyValueSpecification**[14] nodes and relating them to the **ControlAction** through the *object*[15] property. Required parameters that

[10] https://schema.org/SoftwareApplication
[11] https://schema.org/EntryPoint
[12] https://schema.org/actionApplication
[13] https://schema.org/ControlAction
[14] https://schema.org/PropertyValueSpecification
[15] https://schema.org/object

point to content available in the CE, like the video recording the user needs the process to act on, can be specified by adding and relating a **Property**[16] node through the same *object* property.

Together, these **EntryPoint**, **ControlAction**, **PropertyValueSpecification** and **Property** nodes determine what the user pilot will interact with when requesting and controlling a process. This model provides enough information to dynamically generate a process-specific user interface. A user requesting a job through this interface will instantiate the model as a job request which can then be picked up, followed and controlled by the user and by the algorithm process application.

5.1.3 Instance nodes

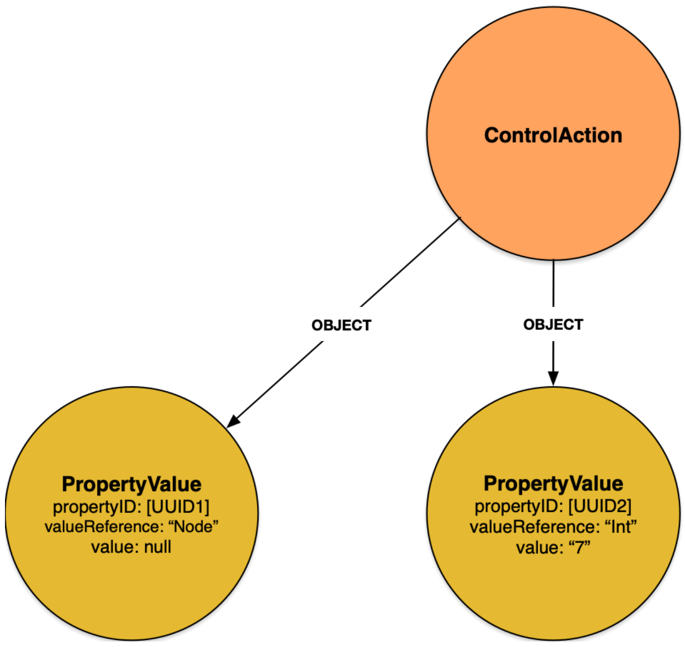


Figure 5.2. Instance nodes schema.

The CE GraphQL interface exposes a predefined mutation (RequestControlAction) that will create a set of nodes based on the 'template' as presented in Section 5.1.2. In effect, the nodes created by this request instantiate a 'job' in the CE database that can now be acted on, followed and updated.

If the RequestControlAction request passes validation, it will create a **ControlAction** node that is a copy of the 'template', plus one or more **PropertyValue**[17] nodes, derived from the template property nodes, that contain the parameters needed to execute the algorithm process. The thus created **ControlAction** serves as the 'job' to be executed, and can now be followed and acted on by both the requester (Component user) and the algorithm maintainer.

[16] https://meta.schema.org/Property
[17] https://schema.org/PropertyValue

5.1.4 Public nodes

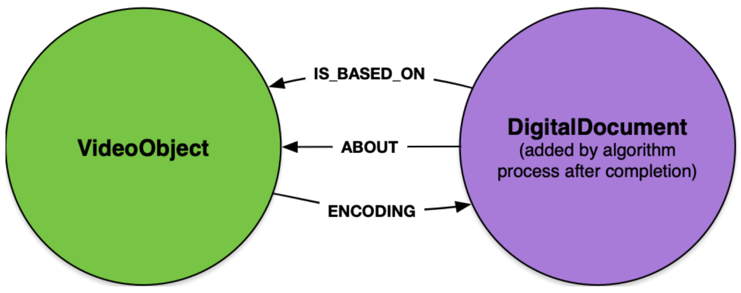


Figure 5.3. Public nodes schema.

The starting point of most algorithm process requests will most likely be one or more (music) content files that are already known in the CE database, or were just uploaded by the user. At the process algorithm request (RequestControlAction), a PropertyValue node was generated that will point at this selected content file reference through the *nodeValue* property.

After picking up and completing the 'job' by, for example, creating a result file at a public location, the algorithm process application needs to create a reference node in the CE database for this result file. It can then relate the ControlAction 'job' node to this result through the *result* property. To allow this result file to turn up in user searches, or offer it to a user who is about to request the same algorithm process on the same source, it is suggested to interlink this new reference to as many relevant nodes as possible. This interlinking is the responsibility of the algorithm process application.

5.2 Communication with the GraphQL Component

5.2.1 Algorithm Subscription

Following the data model presented in Section 5.1, each algorithm must "subscribe" it self to the Contributor Environment. The procedure to do so is summarized as:

- Create a **SoftwareApplication** node in the CE: This node does not correspond to the specific algorithm, but rather to the software/library that hosts the specific algorithm.
- Create an **EntryPoint** node in the CE: The entry point corresponds to a specific algorithm/method to be run.
- Create an *actionApplication* relation between **SoftwareApplication** and **EntryPoint**: This actions defines that this EntryPoint is a part of the **SoftwareApplication** node**.**
- Create a **ControlAction** template node: This **ControlAction** node will be the model for the 'job' created for a specific algorithm process requests. Each request will result in a copy of this **ControlAction** node to be created (instantiated) which will then represent the 'job' that can be acted on and followed.
- Create *potentialAction* relation between **EntryPoint** and (template) **ControlAction:** Relates the **ControlAction** template to the specific **EntryPoint**.
- Create **Property** template node: Corresponds to existing nodes in the CE database, probably referencing a content file at some public repository. These will be the inputs to the WP3 algorithms.

- Create **PropertyValueSpecification**: Corresponds to a scalar parameter (string, number, on/off checkbox) that needs to be given by the user as 'settings' inputs, in order to tune the algorithm process.
- Create object relation between **ControlAction** and **Property**/**PropertyValueSpecification** template nodes respectively.

5.2.2 Algorithm Invoking

Following the data model presented in Section 5.1, each algorithm is invoked by the user pilots, using the RequestControlAction.

- The user pilot makes a RequestControlAction for a specific EntryPoint. The CEapi translates this request by creating a set of nodes on the basis of the **EntryPoint/ControlAction** template and subsequently responds with the thus created **ControlAction**, including its unique *identifier*. With this identifier, the user pilot can then subscribe to the CEapi and and receive a notification each time thus created **ControlAction** is updated.
- From the WP3 program/algorithm perspective, there are two ways to get notified that a new task has to be run:
    - Subscribe to the CEapi on RequestControlAction requests on a specific EntryPoint, through the websocket: This functionality offers the possibility to handle user requests for algorithms in real-time.
    - Frequently check for tasks in the CE: It is possible to query for **ControlAction**s created on the basis of a certain **EntryPoint**. In this way, the software developed under WP3 can check if new tasks have to be run at convenient times or intervals. This functionality offers WP3 software to handle user requests for algorithms in batches.

5.2.3 Algorithm Processing and Monitoring

Since the algorithm has been invoked, it can update the status of **ControlAction** item created during its process until the process is completed:

- Once the algorithm receives the job, it can update the status of the ControlAction node to 'received'.
- While the algorithm is running, it can update the status of the ControlAction node in order to provide more information (e.g. 'running') .
- Once the process has completed, the algorithm process application should write the result to a public location and add a reference to this result in the CE database:
- With the identifier obtained from the response of the result node (e.g. **DigitalDocument**) creation, create a *result* relation between **ControlAction** and **DigitalDocument**:
- When the algorithm process application now updates the **ControlAction** actionStatus to 'complete', the process request response cycle is completed.

5.3 Individual Tools

5.3.1 Overview

The process described above provides a flexible framework for communicating WP3 tasks/jobs with the CE data, and the user pilots. Figure 5.4, shows a general schema that will be adopted for all WP3 components.

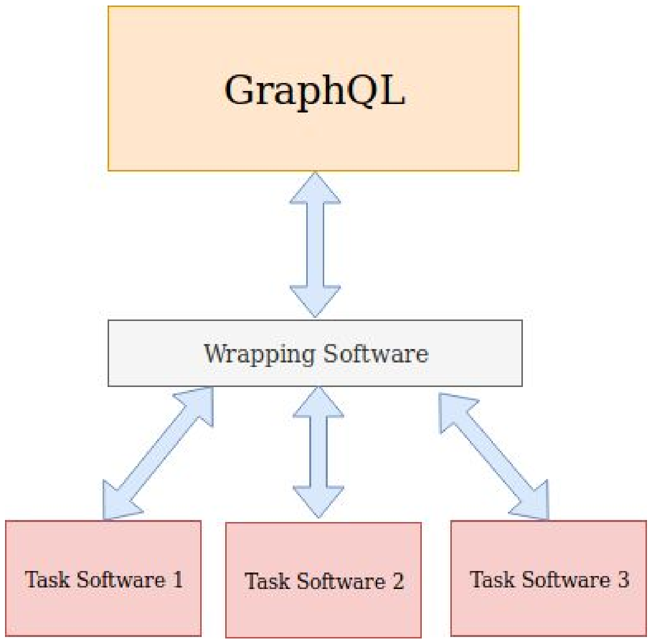


Figure 5.4. Schema of a task specific software organization.

For running a specific task, two different types of components are involved:

- **Wrapping software**: This software is responsible for handling the communication and the requests from/to the GraphQL interface of the CE, and trigger the specific **Task software** needed to run.
- **Task software**: The software that run the actual computations of the task.

In general, these software might be hosted in different servers. This procedure is global (generic) and can be applied to all of the technologies under WP3. In this section, we will provide more details on the WP3 technologies side, e.g. where these technologies will be hosted and run, and where the results will be saved. In the rest of this section, we will dedicate one subsection to each of the Tasks 3.2 - 3.6. Regarding **Task 3.6 Multimodal Cross Linking**, although the corresponding technologies are not directly linked to any of the user pilots (i.e. these algorithm will not be triggered by certain user pilots) but are rather technologies that will be run in the data of the CE outside of the pilots (i.e. linking of CE data to contextual information, external sources such as newspapers, etc), these are still inside the framework of the Trompa Processing Library for reasons of clarity.

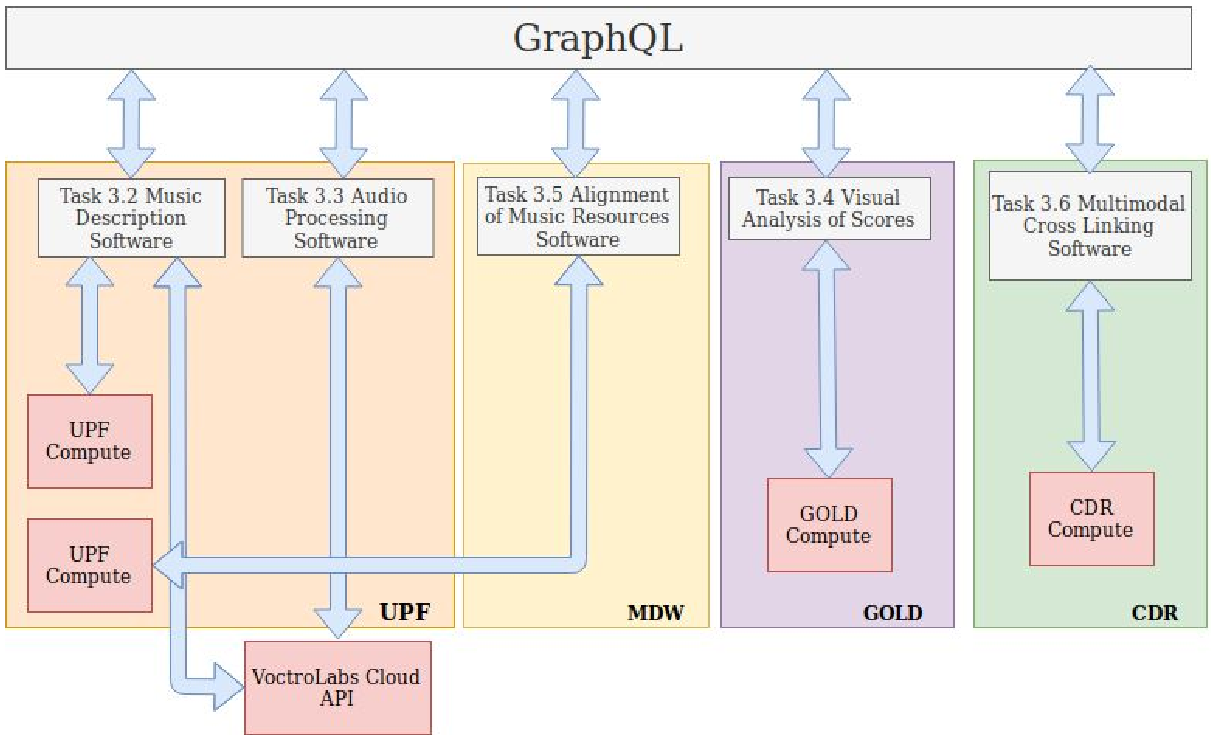

GraphQL
Task 3.2 Music Description Software
Task 3.3 Audio Processing Software
Task 3.5 Alignment of Music Resources Software
Task 3.4 Visual Analysis of Scores
Task 3.6 Multimodal Cross Linking Software
UPF Compute
UPF Compute
GOLD Compute
CDR Compute
VoctroLabs Cloud API
UPF
MDW
GOLD
CDR

Figure 5.5. Overview of the WP3 Application Components

Based on the current status of the integration, an overview of the organization of the software components is illustrated in Figure 5.5. In principle, the intention is to run all the **Wrapping Software** (grey boxes) components in a dedicated server provided by UPF, which are responsible for handling the communication with the GraphQL interface, the job requests, and trigger and invoking the appropriate programs. However, where the actual computations will run depends on the developers of the individual tasks. For instance, regarding Task 3.3 (and some subtasks under Task 3.2) related software, although the software for handling the job requests will run on the UPF server, the synthesis task will run on the Voctro Labs infrastructure; or for Task 3.5 there is the intention to use one of the UPFs computing servers. The same stands for other tasks as well; the intention is to have as many as possible components run in the same hardware as the project progresses. In the rest of this section, we provide some details on the individual tasks and subtasks of WP3.

5.3.1 Music Description

Below are described the technologies under Task 3.2 - Music Description, in correspondence with the Deliverable 3.2 - Audio Description.

5.3.1.1 Low / Medium Audio Descriptors

Low level and medium (tonal and harmonic descriptors) correspond to Section 2.2.1 and 2.2.2 of Deliverable - 3.2 Music Description[18]. These descriptors will be extracted with the use of *Essentia* [1] software. Essentia is a C++ library with Python bindings for audio analysis, synthesis and description. It contains a collection of algorithms that implement standard digital signal processing blocks, a large set of spectral, temporal, tonal and high-level music descriptors, statistical characterization of data.

[18] https://trompamusic.eu/deliverables/TR-D5.3-TROMPA_Processing_Library_v1

Essentia focuses on optimization in terms of robustness, computational speed and low memory usage. After essential is subscribed in the CE, an **EntryPoint** will be created in the CE that corresponds to each type of (low-level/tonal/harmonic) descriptors. A program that listens to the appropriate CE websocket for requests related to that **EntryPoint** and triggers Essentia will be a part of the Trompa Processing Library. Essentia takes as input audio files and will be hosted in UPFs dedicated computer/server for TROMPA related computations. All descriptors can be run in both batch / single file mode. The descriptors will be stored at a UPF server, and will be accessible through public URIs. The output format of the descriptors will be either json or binary files. The parametrization of the algorithm will be done by the corresponding task scientists, in order to provide the best representations suited for TROMPA. Thus, in general, parametrization of the descriptors (i.e. from user pilots, or the users) is not necessary and will not be possible.

5.3.1.2 High Level Audio, Symbolic and Video Descriptors

Regarding high level descriptors that correspond to Rhythm Descriptors, Music Similarity, Emotion Tag Annotation (sub-sections 2.2.3, 2.2.5 and 2.2.6 of Deliverable - 3.2 Music Description), Symbolic Descriptors (Section 2.3) and Video Descriptors (Section 2.4) we will follow the same approach with the one described in the previous subsections, with the only difference that the algorithm software will not be Essentia, but inhouse UPF methods or other open source state-of-the-art methods that will be deployed.

5.3.1.3 Singing Voice Analysis

As depicted in Figure 5.5, the singing analysis technology to be developed in Task 3.2 will be deployed using different algorithms: Voiceful Cloud's *VoDesc, Essentia* and the TROMPA Choir Singing Analysis algorithm.

For the first one, we will integrate Voctro Labs' Voiceful Cloud API, using the *VoDesc* API. The documentation for this Cloud API can be found at[19]. *VoDesc* computes a number of descriptors on singing voice including dynamics (energy), pitch and notes, onsets, and pauses.

For the latter, we will deploy the algorithm to compute choir-singing-specific descriptors developed in Task 3.2. This algorithm requires the score of the performed piece, the result of VoDesc and the score of the piece. Two **EntryPoints** will be created in the CE that correspond to the aforementioned analysis algorithms. Note that Essentia is already assigned EntryPoints (see Section 5.3.1.1) for the low/medium level features. Then, three programs listening to the appropriate CE websocket for requests related to those **EntryPoints** will be part of the Trompa Processing Library. One of them will simply handle the communication between the CE and the Voiceful Cloud service.

These programs will be hosted in UPFs dedicated computer/server for TROMPA related computations; the *VoDesc* algorithm will be (is) hosted in the Voiceful Cloud. The *VoDesc* algorithm takes as input the audio of the singing performance and outputs a JSON-formatted file with the computed descriptors. The same applies for Essentia. The Choir Singing Analysis algorithm will take three nodes as input: the score of the performed piece, the *VoDesc* result, and the result of the Essentia feature extraction. In the case where research designates the use of additional features outside Essentia, there the algorithm might also require the audio of the performance. It will also output a JSON-formatted file with the computed descriptors.

[19] https://cloud.voctrolabs.com/docs/api

The analyses will be able to be run on batch and single file mode. For batch mode, in the case of *VoDesc*, the script running in the UPF server will take care of scheduling the calls to the Voiceful Cloud API, which does not support batch mode. The results of *VoDesc* will be stored in the Voiceful Cloud S3 servers; the descriptors of Essentia and the Choir Singing Synthesis algorithm be stored at a UPF server. In both cases, these results will be accessible through public URIs. The parametrization of the algorithm will be done by the corresponding task scientists, in order to provide the best results suited for TROMPA. Thus, in general, parametrization of the descriptors (i.e. from user pilots, or the users) will not be possible.

5.3.2 Audio Processing

The singing synthesis will be integrated using Voctro Labs' Voiceful Cloud service. The motivation for using this platform is twofold: first, it is an existing working solution for singing synthesis that already stores its results in S3 servers, publicly accessible through URLs to which the CE can store references; second, it facilitates the immediate adaptation of the developed technologies in other uses cases for exploitation that may arise beyond the project.

Since the synthesis will occur in the Voiceful Cloud, the scripts that will run in the UPF dedicated server (Figure 5.5) will just take care of the communication with the CE. The will subscribe to the relevant nodes in the CE following the procedure explained in 5.2.1, and will call the Voiceful Cloud service during execution. Example code snippets of how this cloud service is used can be found online[20]. This Cloud API service will be extended for TROMPA and will consist of two main tasks:

- ❖ Deploying the **new models** in the cloud servers. As detailed in **Deliverable 3.3 - Audio Processing**[21], choir singing synthesis has specific requirements beyond those of solo singing synthesis. During the project, new synthesis models will be created from new recordings for multiple languages. These new models will be made accessible through the cloud service.
- ❖ Adapting the API to support the choir case. Currently, the *VoSynth* API accepts monophonic input scores in MusicXML format, as well as in other specific .txt and .json formats for synthesis. For the choir use case, the API will be adapted in order to accept also MEI scores and to support polyphonic material, i.e. receiving a score containing all the voices to be synthesized and outputting one audio file per voice.

As depicted in Figure 5.5, the singing synthesis technology developed in Task 3.3 will be integrated into the TROMPA processing library through Voctro Labs' Voiceful Cloud API, using the *VoSynth* API. The documentation for this Cloud API can be found online[22].

An EntryPoint will be created in the CE that corresponds to the singing synthesis. A program that listens to the appropriate CE websocket for requests related to that EntryPoint and calls the Voiceful Cloud to perform the synthesis will be a part of the Trompa Processing Library. The program taking care of the communication with the CE will be hosted in UPFs dedicated computer/server for TROMPA related computations; the synthesis algorithms will be hosted in the Voiceful Cloud. The synthesis algorithm takes as input the score and outputs as many audios as voices are contained in the score (e.g. one audio for Soprano, one for Alto, one for Tenor and one for Bass). The synthesis will be able to be run on batch and single file mode. For batch mode, the script running in the UPF

[20] https://www.voiceful.io/developers
[21] https://trompamusic.eu/deliverables/TR-D3.3-Audio_Processing_v1
[22] https://cloud.voctrolabs.com/docs/api

server will take care of scheduling the calls to the Voiceful Cloud API, which does not support batch mode.

The results will be stored in the Voiceful Cloud S3 servers, accessible through public URIs. The parametrization of the algorithm will be done by the corresponding task scientists, in order to provide the best results suited for TROMPA. Thus, in general, parametrization of the descriptors (i.e. from user pilots, or the users) will not be possible. More concretely, we will make use of the Voiceful Cloud "Presets" capability. This allows to modify parameters from the cloud management console while keeping the same call from the client side.

5.3.3 Visual Analysis of Scores

A machine-readable representation of a musical score for use in a project such as TROMPA can be generated in one of three basic ways: manual encoding, optical music recognition (OMR) or conversion from an existing representation. Manual encoding (using a score editing program such as Finale, Sibelius, Dorico or MuseScore) is time-consuming and requires considerable expertise, but is useful for scholarly transcriptions or performing editions of individual works; in general one expects the quality to be high. Conversion methods depend on the existence of files in formats such as MIDI, which can be 'translated' into a suitable data-representation under certain heuristic assumptions; these assumptions cannot in general be relied upon as they concern the addition of information that may not be present in the source format. For example, MIDI does not represent barlines/measures (all note-events are merely time-stamped) and no pitch-spelling information is present (there is no distinction in MIDI between an A# and a Bb, as these are played by the same key on an electronic keyboard).

For several purposes within TROMPA involving the on-screen display of scores we shall be using OMR technology to extract complete or partial machine-readable score representations in the MEI format from scores saved as PDF or other graphical formats. While there are several commercial programs that might be suitable for very clean, modern engraved scores, we shall be using public-domain scores often printed more than a century ago and which can present significant challenges in this respect. OMR software is by no means reliable, and its performance is highly dependent on the condition of the original materials, the quality of photography and aspects of digitisation beyond our control. For this reason we choose to work with open-source programs which can be modified or adapted to our purposes, which can be run in batch-mode over large collections of music, and which save their results in a form (such as MusicXML) which can be easily converted to MEI for use within TROMPA.

OMR is also dependent on the style of the music notation. In general, it is not possible to use it on manuscript scores at all, and only certain types of historical printed scores are useable. For standard modern musical notation we shall be using the open-source program Audiveris,[23] which is under continuous development and explicitly focussed on the demands of the IMSLP resource which we also make use of. Audiveris is generally reliably accurate with good-quality scores in good condition and well photographed/digitised. With poorer-quality printing or photography, like all OMR programs, its accuracy declines markedly. However, for some purposes in TROMPA we do not need full OMR transcription into MEI; a partial representation is sometimes sufficient. For example, in order to align a score with audio (so that pages can be 'turned' on-screen as the music plays) it may only be necessary to locate all the barlines in the score and record links to time-offsets within the

[23] https://github.com/Audiveris/audiveris

audio stream; first experiments suggest that Audiveris is very reliable for this, even when image-quality is poor.

For the music-scholars' use case within TROMPA, we shall also be working with vocal music from the 16th century. While much of this is available in modern editions (many of these are in IMSLP), we shall also work with facsimile images of the original sources (which may include music not available in modern editions). These were in almost every case in the form of individual (monophonic) part-books for a single singer, printed from type, which requires a different approach than engraved modern notation. We shall thus use the specialist software Aruspix,[24] with which the GOLD team has long-term experience, for OMR on resources such as the British Library's Early Music Online. Again, the accuracy of recognition is very dependent on the condition of the original materials, but the output of Aruspix is mostly reliable, at least in terms of relative pitch, which means that we can use the MEI output to generate indexes which can be used to retrieve pages of the original source from a given score or fragment thereof, even in the presence of a considerable degree of OMR error. This novel music-retrieval paradigm has great potential in musicology, and need not be restricted to early music, as it can be used just as easily with the output of Audiveris, so we shall be experimenting with the possibility of content-based indexing of all TROMPA scores. Like Audiveris, Aruspix can be run as a batch program, but it is also fast enough to be used on the fly on user-uploaded images of 16c music pages; the uploading and recognition of a page, query-extraction and search of a database of 32,000 pages in the EMO test collection takes under 10 seconds, including network transfers.[Crawford, T., Badkobeh, G. and Lewis, D. (2018) Searching Page-Images of Early Music Scanned with OMR: A Scalable Solution using Minimal Absent Words. ISMIR 2018]

Similar to other WP3 technologies, each of the software that will be used will be assigned to an **EntryPoint.** The wrapper software (Figure 5.4) will be (possibly) hosted on the UPF server. Regarding the computations components, the infrastructure that will be hosted is to be determined in the future with respect to the computation demands, which depend on the use cases that involve OMR.

With both the early music and classical repertoire for which these software packages will be used within TROMPA, there is no need for real-time or on-the-fly recognition (apart from the uploaded-image search mentioned above, which will be handled externally to TROMPA via an API managed by Swiss RISM), and recognition can be run in the background as a batch process. An important issue is the management of the associated image and source metadata, which we anticipate will be handled by the Contributor Environment. The output and extracted features/indexes will be stored centrally under the management of the Contributor Environment and using the affordances of Linked Data, which further allow us to include provenance information such as the program used, details of image preprocessing (such as splitting of double-page spreads into individual pages) and any non-default parameter settings used in the recognition process.

5.3.4 Alignment of Music Resources

Alignment of Musical Resources (Task 3.5) concerns the interlinking of multimodal representations of a musical expression at different levels of granularity. At current stage of development (M12), modalities involve musical score encodings (encoded using, or converted to, the MEI format); and MIDI-encoded performance renditions. Audio-to-audio alignments will also be targeted in future development. Various algorithms and software packages to accomplish this task are described in

[24] http://www.aruspix.net

Deliverable 3.5 - Multimodal Music information Alignment[25]. For present development, we are focussing our efforts on the MAPS tool ("Matcher for Alignment of Performance and Score") under ongoing in-house development at MDW, which has the advantage of natively supporting MEI.

MAPS will be called through a wrapper layer ("MAPS wrapper") responsible for accepting new jobs (e.g., performances) from TROMPA users, registering these within the CE, and spawning MAPS instances to complete the jobs. MAPS instances are processor intensive and should thus be deployed on a dedicated high-performance server - potentially at UPF. The MAPS wrapper is a comparatively light-weight web service that could be deployed separately, e.g. at MDW. These decisions are as yet under discussion and will be settled in due course. The output generated by each job (i.e., RDF triples instantiating the alignment data model, see D3.5) will be stored in a Linked Data Platform (LDP) server at MDW, referred to by URI (as DigitalDocuments) by the CE.

The input of the MAPS in the core encoding (only required once per MEI file), the MIDI recording (one per each performance; either as a MIDI file for offline processing, or as a live MIDI stream for online processing), and the output is the outcome of the alignment process (RDF triples instantiating the alignment data model) are stored in a Linked Data Platform (LDP) server at MDW, referred to by URI (as DigitalDocuments) by the CE. Alignments are generated on a note-level; higher-granularity alignments are automatically determined from this by mapping upward from notes within the MEI hierarchy. Additionally, each alignment is associated with a confidence measure reflecting the coefficient of variation (a measure of "peakiness") of the MAPS hidden Markov model state probability distribution; it varies from 0% (a particular MIDI event can be associated to any MEI note with equal likelihood) to 100% (the MIDI event is associated with a particular MEI note with perfect certainty).

MAPS supports both offline processing using MIDI files, and online processing using streamed MIDI events. Offline processing is more reliable but requires long processing times which may impact usability (e.g., an 11-minute performance of Beethoven's 32 Variations in C Minor - WoO 80 takes around 7 minutes to compute on a desktop workstation; though improvements are expected with further development, and on machines with more CPU cores). Online processing returns results in real-time (or just after a performance rendition is completed, depending on user requirements), with potentially reduced reliability. Descriptors generated by the alignment process will be stored in the LDP server hosted by MDW, and referred to (by URI) by the CE.

Regarding the parameterization of the tool, users must provide a score encoding (MEI file). In batch mode, a MIDI file must be provided; in online mode, MIDI events are streamed from an in-situ device colocated with the performance (e.g. iPad connected to a MIDI piano). Further parameters can be specified to configure the hidden Markov model; choose and configure a tempo model; and configure pitch profiles.

Regarding error handling, non-valid MEI files will be detected and cause an exception to be raised; this will be intercepted by the MAPS wrapper and dealt with appropriately (by updating the CE node to an error state). Non-valid MIDI files will be similarly handled in batch mode. Dropped network connections during MIDI streaming in online mode will be handled through a time-out mechanism that will preemptively complete the job (publishing the processed alignments reflecting the MIDI events received before the connection ended).

[25] https://trompamusic.eu/deliverables/TR-D3.5-Multimodal_Music_Information_Alignment_v1

5.3.5 Multimodal Cross Linking

Multimodal Cross Linking comprises of two different pieces of software:

- ❖ **Authority registration and linking**: This software has a user interface to create and describe web-resources and the relations between entities and resource types. For example Wikidata will be registered as authority that provides Person Entities, Musical Work Entities and other entities that will be stored in the CE. Links to other resources will be described like **SameAs** if the same entity is linked to another resource.
- ❖ **Web scraper**: The web scraper will retrieve entity information from the in the CE stored web-resource and finds links to other registered authorities. It will save the referenced entities in the CE.

The user interface (Authority registration and linking) interacts with contributors and the CE. On the other hand, the web scraper interacts between the CE and the web resources of registered authorities. The multimodal cross linking will have use cases where the linked data can be used as background reference or where data can be enriched for the end user. This is primarily with cases for music scholars and music enthusiasts.

Both pieces of software will run as micro services in a cloud environment under subscription of Trompa. The authority registration and linking software will register user input from a web-interface and store the data in the CE. The web scraper takes data from web-resources of registered authorities and saves the linked data in the CE.

Regarding error handling, the web service for the user interface for the authority registration should generate error messages for the user so technical support can be used to resolve the problems. For the web scraper, an e-mail report must be generated that will be sent to the administrators of the application.

6. Conclusion

In this deliverable we described in detail the TROMPA Processing Library. Rather than being a single piece of software, TROMPA Processing Library is a collection of Contributor Environment functionalities that allow the interaction of WP3 components with the CE data, the different software components that correspond to the various WP3 tasks, and the organization amongst them. We presented the details of the mechanism for storing and retrieving data object in the CE, assigning specific processing tasks for the data, and the overall software and partially hardware organization of the different components. The proposed architecture is flexible and easy to adapt to the project needs, as this progresses.

For the next period of the project, we plan to develop the TPL as follows:

- ❖ **Preliminary Development** (M13 - 18): By the end of this period we will test all of the individual components of WP3 and validate the correct communication with the CE: assignment of tasks from the CE, execution of tasks, storage of data.
- ❖ **First Working Version** (M19-M22): By the end of this period we will provide a first full version of the TPL. This will be delivered 2 months prior to M24 and MS3, where the first version of the working prototypes of the pilots will be delivered in order to facilitate the development of the pilots.
- ❖ **Incorporation of Latest WP3 Components** (M23-24): TPL will incorporate the final versions of for Task 3.3 - Audio Processing and Task 3.5 - T3.5 Alignment of musical resources.

- ❖ **Incorporation of Final WP3 Components** (M25-M30): TPL will incorporate the final versions of for Task 3.4 - Visual Analysis of Scanned Scores and Task 3.6 - Multimodal Cross Linking
- ❖ **Final Version** (M31-M34): Final adaptations/debugging. There will be an effort to have most of the components centralised for ensuring sustainability after the end of the project.

7. References

7.1 Written references

[1] Bogdanov, D., Wack N., Gómez E., Gulati S., Herrera P., Mayor O., et al. (2013). ESSENTIA: an Audio Analysis Library for Music Information Retrieval. International Society for Music Information Retrieval Conference (ISMIR'13). 493-498.

[2] Crawford, T., Badkobeh, G. and Lewis, D. (2018) Searching Page-Images of Early Music Scanned with OMR: A Scalable Solution using Minimal Absent Words. ISMIR 2018

7.2 List of abbreviations

Partner	Description
UPF	University Pompeu Fabra
IMSLP	International Music Score Library Project
GOLD	Goldsmiths University of London
MDW	University of Music and Performing Arts Vienna
CDR	Centrale Discotheek Rotterdam
Abbreviation	**Description**
TPL	Trompa Processing Library
CE	Contributor Environment
CEapi	Contributor Environment API
UI	User Interface
PCP	Pitch Class Profiles
MEI	Music Encoding Initiative
MAPS	Matcher for Alignment of Performance and Score

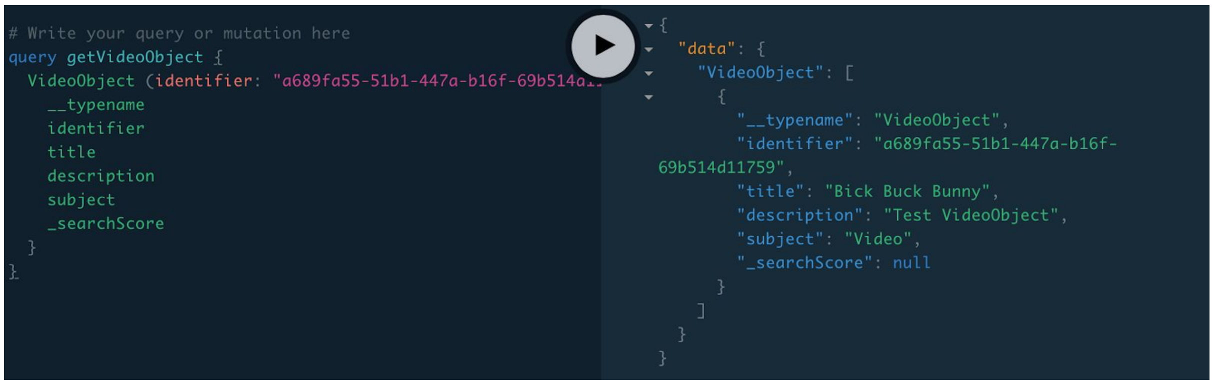
GUI	Graphical User Interface
OMR	Optical Music Recognition

Annex A - Query the Contributor Environment

The contents of this Annex are part of the ongoing working document CE API Manual, which will be frequently updated as the project progresses. At a later stage of the project, the CE API Manual will be publicly available.

1. Queries

1.1 Simple query for one node



```
# Write your query or mutation here
query getVideoObject {
  VideoObject (identifier: "a689fa55-51b1-447a-b16f-69b514a
    __typename
    identifier
    title
    description
    subject
    _searchScore
  }
}
```

```
{
  "data": {
    "VideoObject": [
      {
        "__typename": "VideoObject",
        "identifier": "a689fa55-51b1-447a-b16f-
69b514d11759",
        "title": "Bick Buck Bunny",
        "description": "Test VideoObject",
        "subject": "Video",
        "_searchScore": null
      }
    ]
  }
}
```

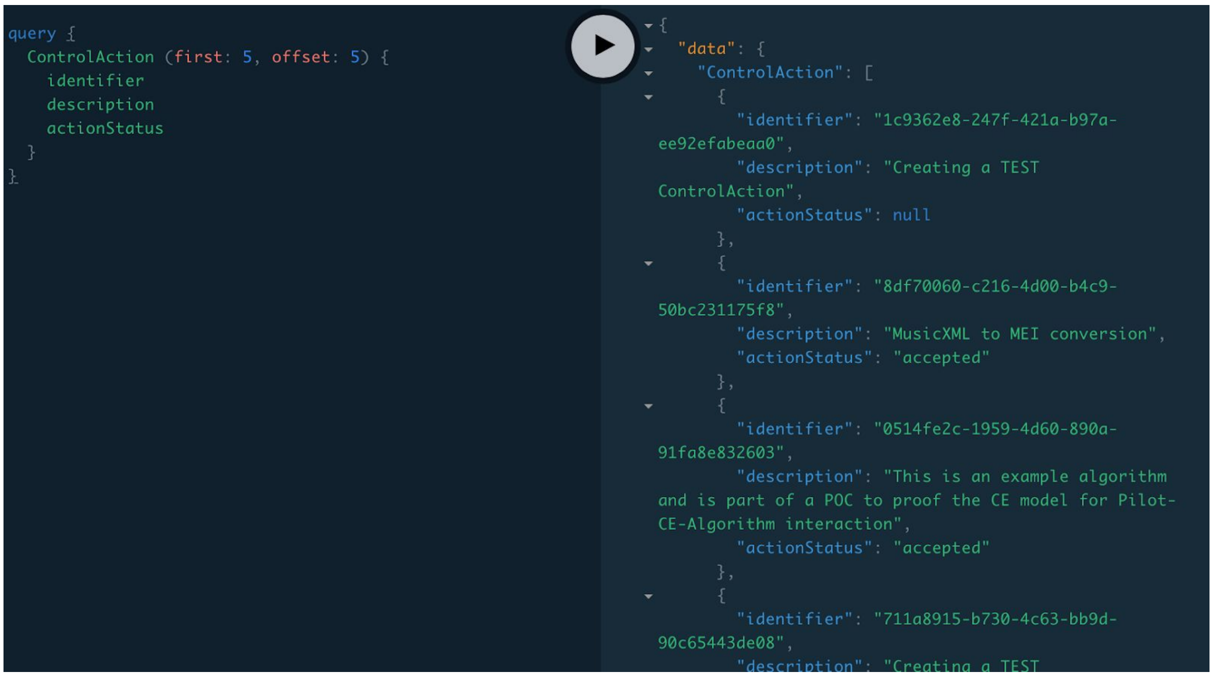
A query starts with the phrase 'query' and typically consists of:

- ❖ The name of the query (optional)
- ❖ Type of entity for which is queried
- ❖ Conditions (optional)
- ❖ List of properties to be included in the response

The result typically consists of json object containing:

- ❖ The "data" object with the result(s)
- ❖ The name of the query responded to
- ❖ The actual data, corresponding to the list of properties to be included

1.2 Simple query for multiple nodes

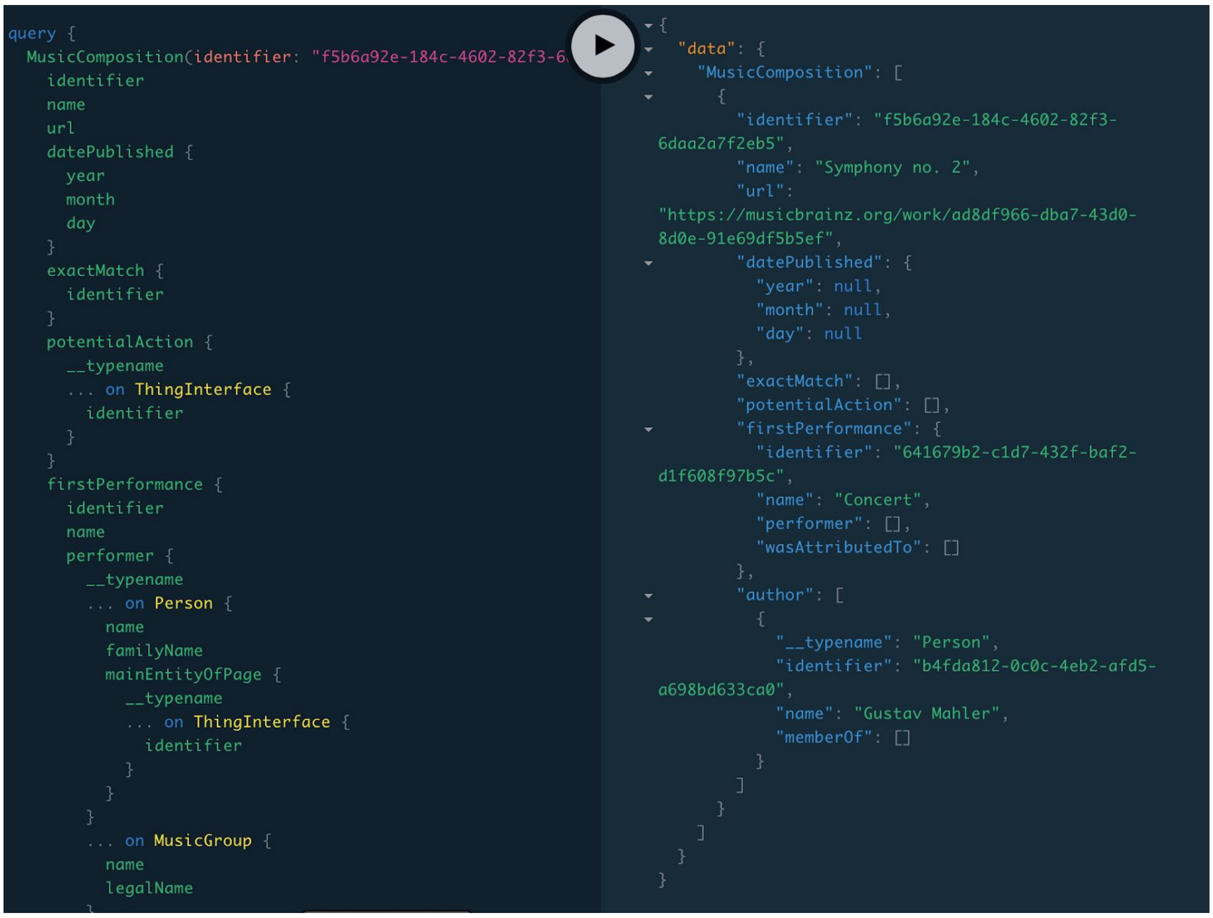


```
query {
  ControlAction (first: 5, offset: 5) {
    identifier
    description
    actionStatus
  }
}
```

```
{
  "data": {
    "ControlAction": [
      {
        "identifier": "1c9362e8-247f-421a-b97a-
ee92efabeaa0",
        "description": "Creating a TEST
ControlAction",
        "actionStatus": null
      },
      {
        "identifier": "8df70060-c216-4d00-b4c9-
50bc231175f8",
        "description": "MusicXML to MEI conversion",
        "actionStatus": "accepted"
      },
      {
        "identifier": "0514fe2c-1959-4d60-890a-
91fa8e832603",
        "description": "This is an example algorithm
and is part of a POC to proof the CE model for Pilot-
CE-Algorithm interaction",
        "actionStatus": "accepted"
      },
      {
        "identifier": "711a8915-b730-4c63-bb9d-
90c65443de08",
        "description": "Creating a TEST
```

By passing first / offset parameters, the results can be paginated

1.3 Complex query



```
query {
  MusicComposition(identifier: "f5b6a92e-184c-4602-82f3-6
    identifier
    name
    url
    datePublished {
      year
      month
      day
    }
    exactMatch {
      identifier
    }
    potentialAction {
      __typename
      ... on ThingInterface {
        identifier
      }
    }
    firstPerformance {
      identifier
      name
      performer {
        __typename
        ... on Person {
          name
          familyName
          mainEntityOfPage {
            __typename
            ... on ThingInterface {
              identifier
            }
          }
        }
        ... on MusicGroup {
          name
          legalName
```

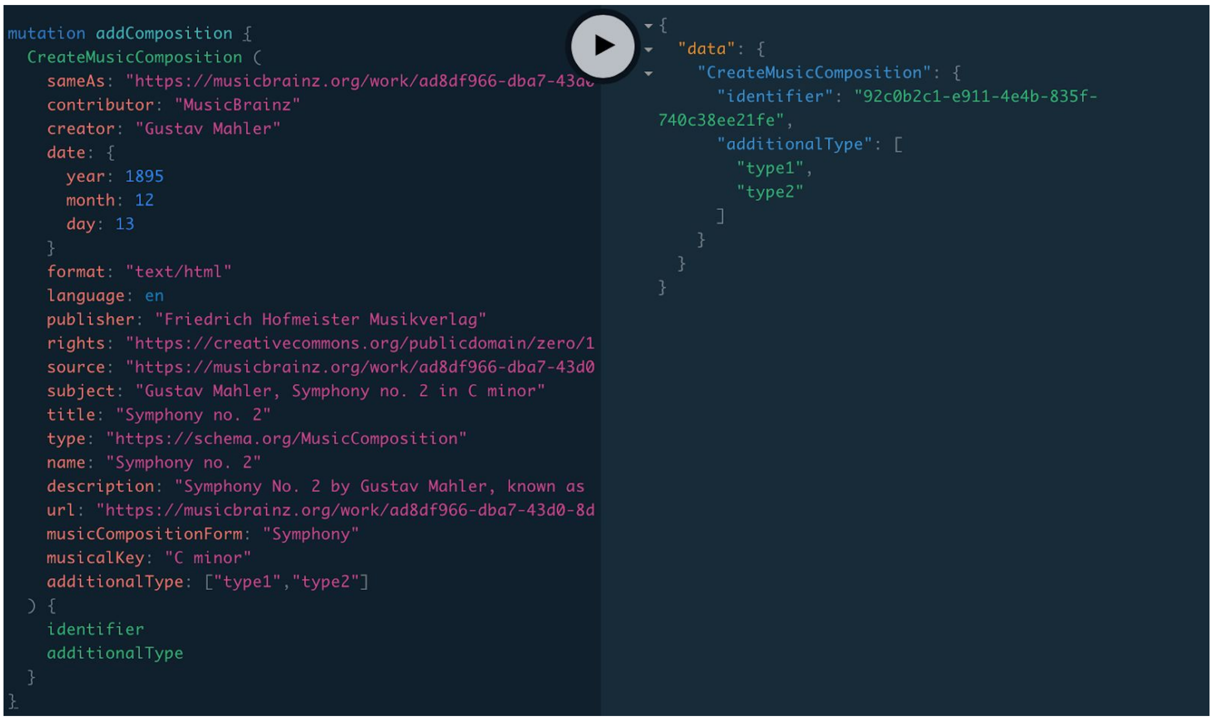
```
{
  "data": {
    "MusicComposition": [
      {
        "identifier": "f5b6a92e-184c-4602-82f3-
6daa2a7f2eb5",
        "name": "Symphony no. 2",
        "url":
"https://musicbrainz.org/work/ad8df966-dba7-43d0-
8d0e-91e69df5b5ef",
        "datePublished": {
          "year": null,
          "month": null,
          "day": null
        },
        "exactMatch": [],
        "potentialAction": [],
        "firstPerformance": {
          "identifier": "641679b2-c1d7-432f-baf2-
d1f608f97b5c",
          "name": "Concert",
          "performer": [],
          "wasAttributedTo": []
        },
        "author": [
          {
            "__typename": "Person",
            "identifier": "b4fda812-0c0c-4eb2-afd5-
a698bd633ca0",
            "name": "Gustav Mahler",
            "memberOf": []
          }
        ]
      }
    ]
  }
}
```

For properties containing nodes, the request body needs to create a deeper property list for that node. In most cases these deeper nodes can be of multiple types. For each expected node type, an `... on [type] {}` property list needs to be included. Thankfully, the CE-api interface suggests options and validates the query in real time.

2. Mutations

Mutations are queries that add, update or remove data in the database.

2.1 Creating a node



```
mutation addComposition {
  CreateMusicComposition (
    sameAs: "https://musicbrainz.org/work/ad8df966-dba7-43a
    contributor: "MusicBrainz"
    creator: "Gustav Mahler"
    date: {
      year: 1895
      month: 12
      day: 13
    }
    format: "text/html"
    language: en
    publisher: "Friedrich Hofmeister Musikverlag"
    rights: "https://creativecommons.org/publicdomain/zero/1
    source: "https://musicbrainz.org/work/ad8df966-dba7-43d0
    subject: "Gustav Mahler, Symphony no. 2 in C minor"
    title: "Symphony no. 2"
    type: "https://schema.org/MusicComposition"
    name: "Symphony no. 2"
    description: "Symphony No. 2 by Gustav Mahler, known as
    url: "https://musicbrainz.org/work/ad8df966-dba7-43d0-8d
    musicCompositionForm: "Symphony"
    musicalKey: "C minor"
    additionalType: ["type1","type2"]
  ) {
    identifier
    additionalType
  }
}
```

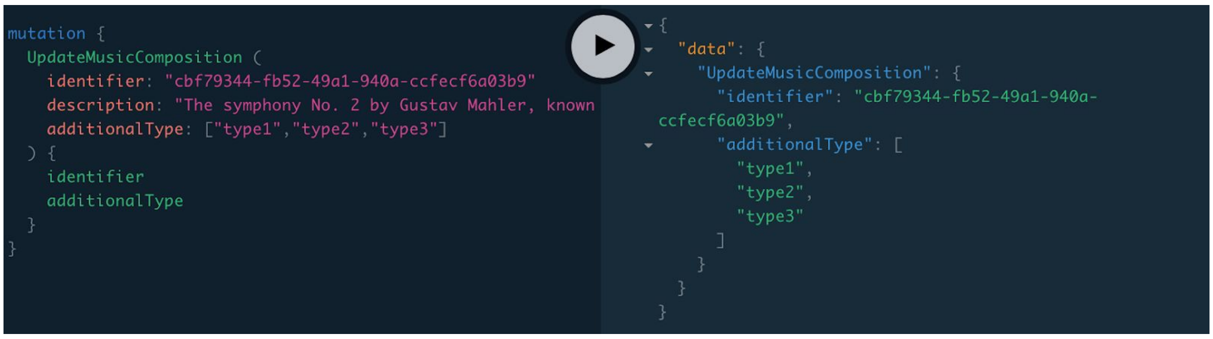
```
{
  "data": {
    "CreateMusicComposition": {
      "identifier": "92c0b2c1-e911-4e4b-835f-
740c38ee21fe",
      "additionalType": [
        "type1",
        "type2"
      ]
    }
  }
}
```

A create mutation typically consists of:

- ❖ Between brackets, a list of scalar parameters that correspond to the type properties for which the value needs to be set. Properties can also contain arrays of scalars.
- ❖ Between curly braces, a list of properties to be returned once the node is created

What cannot be passed as data to be created is a related node, either new or existing. The exception are properties containing 'datetime' type data. The Neo4j database has a number of scalar datetime types. As we use the Apollo library, we need to pass such a date as an object.

2.2 Updating a node

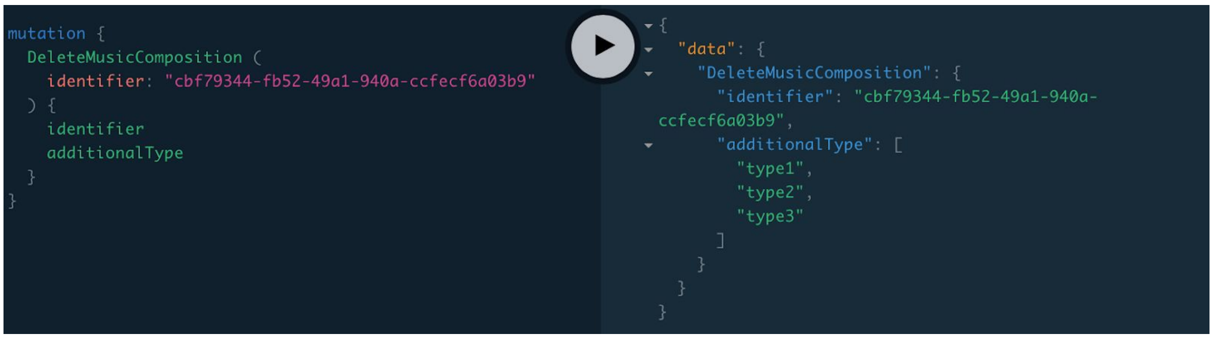


```
mutation {
  UpdateMusicComposition (
    identifier: "cbf79344-fb52-49a1-940a-ccfecf6a03b9"
    description: "The symphony No. 2 by Gustav Mahler, known
    additionalType: ["type1","type2","type3"]
  ) {
    identifier
    additionalType
  }
}
```

```
{
  "data": {
    "UpdateMusicComposition": {
      "identifier": "cbf79344-fb52-49a1-940a-
ccfecf6a03b9",
      "additionalType": [
        "type1",
        "type2",
        "type3"
      ]
    }
  }
}
```

The update query is much like the Create query, with the difference that the identifier needs to be passed along with the update parameters. Properties that are left out will not be updated. When updating an array value, the full array needs to be passed: elements missing from the update data will be removed.

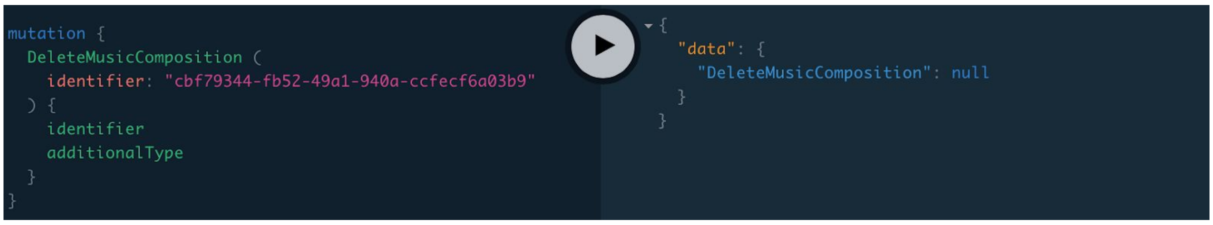
2.3 Deleting a node



```
mutation {
  DeleteMusicComposition (
    identifier: "cbf79344-fb52-49a1-940a-ccfecf6a03b9"
  ) {
    identifier
    additionalType
  }
}
```

```
{
  "data": {
    "DeleteMusicComposition": {
      "identifier": "cbf79344-fb52-49a1-940a-
ccfecf6a03b9",
      "additionalType": [
        "type1",
        "type2",
        "type3"
      ]
    }
  }
}
```

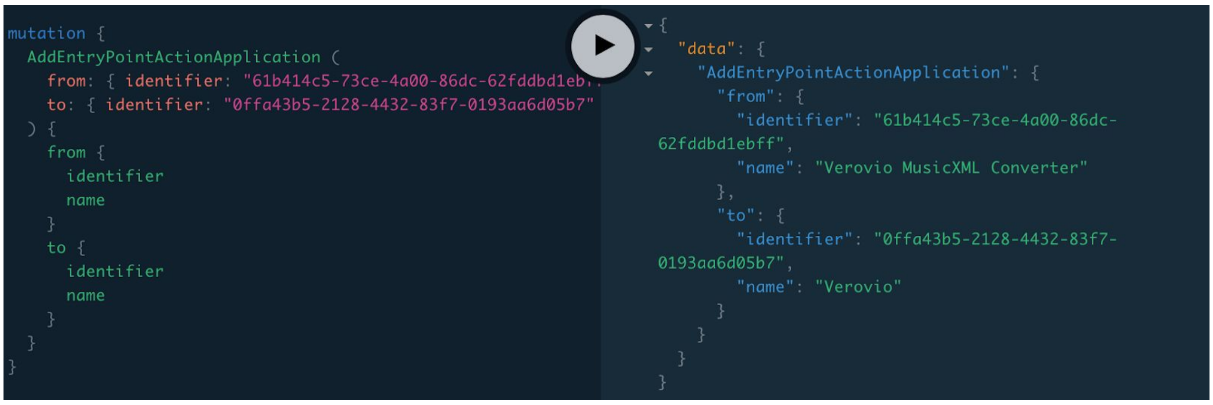
When deleting a node, only the identifier can be passed as a parameter. When a node gets deleted, all its incoming and outgoing relations to other nodes will also be deleted. The response will contain the data of just before the node was deleted. Running the same delete query again would yield an empty result, as there was no more node to delete:



```
mutation {
  DeleteMusicComposition (
    identifier: "cbf79344-fb52-49a1-940a-ccfecf6a03b9"
  ) {
    identifier
    additionalType
  }
}
```

```
{
  "data": {
    "DeleteMusicComposition": null
  }
}
```

2.4 Add a relation between nodes (primitive types)



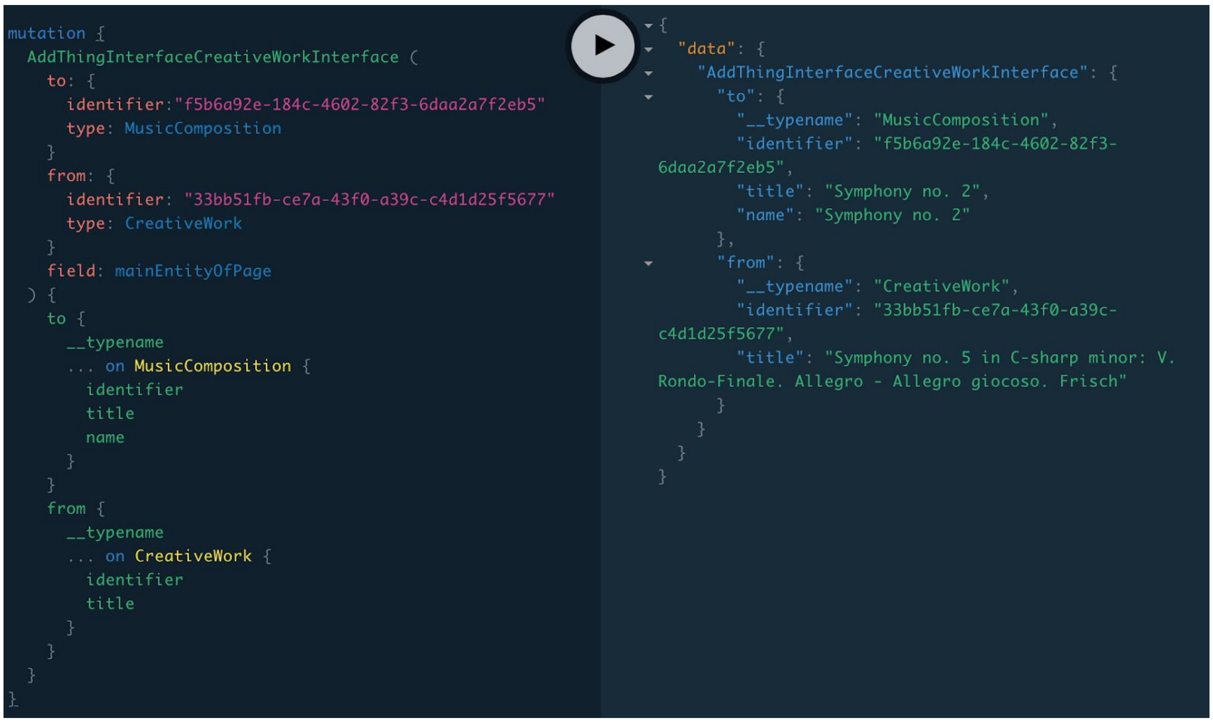
```
mutation {
  AddEntryPointActionApplication (
    from: { identifier: "61b414c5-73ce-4a00-86dc-62fddbd1eb
    to: { identifier: "0ffa43b5-2128-4432-83f7-0193aa6d05b7"
  ) {
    from {
      identifier
      name
    }
    to {
      identifier
      name
    }
  }
}
```

```
{
  "data": {
    "AddEntryPointActionApplication": {
      "from": {
        "identifier": "61b414c5-73ce-4a00-86dc-
62fddbd1ebff",
        "name": "Verovio MusicXML Converter"
      },
      "to": {
        "identifier": "0ffa43b5-2128-4432-83f7-
0193aa6d05b7",
        "name": "Verovio"
      }
    }
  }
}
```

For each relation, which is a property containing another node, there is a dedicated mutation query. Consult the schema to find the right relation mutation. The GraphQL interface has autosuggestion and detects invalid queries.

The mutation create query for a relation between 2 primitive types only needs the identifiers of both nodes as parameters, contained in ‘from’ and ‘to’ objects. It is possible to create multiple relations of the same type between the same nodes.

2.5 Add a relation between nodes (Interfaced or Unioned types)

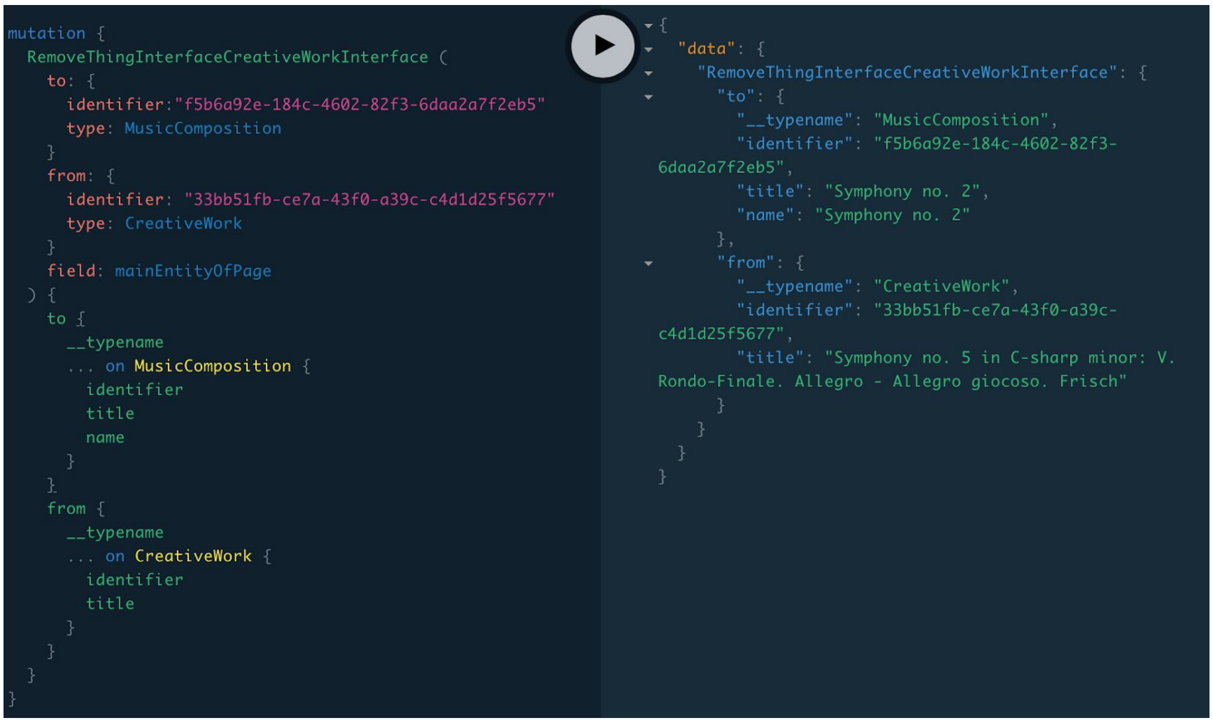


```
mutation {
  AddThingInterfaceCreativeWorkInterface (
    to: {
      identifier:"f5b6a92e-184c-4602-82f3-6daa2a7f2eb5"
      type: MusicComposition
    }
    from: {
      identifier: "33bb51fb-ce7a-43f0-a39c-c4d1d25f5677"
      type: CreativeWork
    }
    field: mainEntityOfPage
  ) {
    to {
      __typename
      ... on MusicComposition {
        identifier
        title
        name
      }
    }
    from {
      __typename
      ... on CreativeWork {
        identifier
        title
      }
    }
  }
}
```

```
{
  "data": {
    "AddThingInterfaceCreativeWorkInterface": {
      "to": {
        "__typename": "MusicComposition",
        "identifier": "f5b6a92e-184c-4602-82f3-
6daa2a7f2eb5",
        "title": "Symphony no. 2",
        "name": "Symphony no. 2"
      },
      "from": {
        "__typename": "CreativeWork",
        "identifier": "33bb51fb-ce7a-43f0-a39c-
c4d1d25f5677",
        "title": "Symphony no. 5 in C-sharp minor: V.
Rondo-Finale. Allegro - Allegro giocoso. Frisch"
      }
    }
  }
}
```

To create a relation that expresses a property that can contain different node types (relation to an Interface or a Union), a number of custom queries were created. Consult the schema to find the right relation query. Depending on the query, the ‘from’ and ‘to’ bodies require an indication of the primitive type of the parent and target node. If the parent node type has several properties that express relations between Interfaced or Unioned types, it is also necessary to include the ‘field’ parameter that indicates which parent property is expressed by the relation.

2.6 Remove a relation between nodes



```
mutation {
  RemoveThingInterfaceCreativeWorkInterface (
    to: {
      identifier:"f5b6a92e-184c-4602-82f3-6daa2a7f2eb5"
      type: MusicComposition
    }
    from: {
      identifier: "33bb51fb-ce7a-43f0-a39c-c4d1d25f5677"
      type: CreativeWork
    }
    field: mainEntityOfPage
  ) {
    to {
      __typename
      ... on MusicComposition {
        identifier
        title
        name
      }
    }
    from {
      __typename
      ... on CreativeWork {
        identifier
        title
      }
    }
  }
}
```

```
{
  "data": {
    "RemoveThingInterfaceCreativeWorkInterface": {
      "to": {
        "__typename": "MusicComposition",
        "identifier": "f5b6a92e-184c-4602-82f3-
6daa2a7f2eb5",
        "title": "Symphony no. 2",
        "name": "Symphony no. 2"
      },
      "from": {
        "__typename": "CreativeWork",
        "identifier": "33bb51fb-ce7a-43f0-a39c-
c4d1d25f5677",
        "title": "Symphony no. 5 in C-sharp minor: V.
Rondo-Finale. Allegro - Allegro giocoso. Frisch"
      }
    }
  }
}
```

But for the Query name, the Removal query for a relation is identical to the Create query. A Remove query removes all the relations of the same type between the indicated nodes .

Annex B - Verovio Converter Example

An example on how to use the Verovio[26] software as a component of the TROMPA Processing Library is provided. Verovio is a fast, portable and lightweight open-source library for engraving Music Encoding Initiative (MEI) music scores into SVG. In the this Github repository[27] here is a working example of Verovio MusicXML->MEI conversion on eligible data contained in the CE database.

[26] https://www.verovio.org/index.xhtml
[27] https://github.com/trompamusic/ce-digital-score-edition-component