

# TROMPA

TROMPA: Towards Richer Online Music Public-domain Archives

## Deliverable 2.3

### Technical requirements and integration

Grant Agreement nr	770376
Project runtime	May 2018 - April 2021
Document Reference	D2.3_Technical_requirement_and_integration_v1
Work Package	WP2 - Project Coordination
Deliverable Type	Report
Dissemination Level	CO
Document due date	31 October 2019
Date of submission	31 October 2019
Leader	5 - VD
Contact Person	Bauke Freiburg (bauke@videodock.com)
Authors	Wim Klerkx (VD), Aggelos Gkiokas (UPF), Bauke Freiburg (VD), Christiaan Scheermeijer (VD), Alastair Porter (UPF), David Weigl (MDW)
Reviewers	Vladimir Viro (PN)

D2.3\_Technical\_requirement\_and\_integration\_v1

## Executive Summary

This deliverable is the technical requirements and integration report and specifies the strategies for technical integration of data generated by different technologies. The initial version includes the integration guidelines to be followed during the project that will be updated in the final version. The document is created by merging two earlier working documents (CE API Manual and CE Import Guidelines). The document is written for a technical audience and should provide practical information and guidelines for developers and researchers on how to integrate with the Contributor Environment (CE). It is expected that the document is updated and improved after its initial submission. The goal would be that at the end of the project, third parties with little background information on the project understand how they can integrate with the CE based on this technical documentation.

Section 2 presents 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. 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. All nodes in the CE will be either one of the eight base types from schema.org, namely CreativeWork, Event, Person, Organization, Product, Action, Place and Intangible. These eight types are extended from schema.org's ultimate base type [Thing](#) which has a number of properties which are available for all types, and thus all nodes that reside in the CE. These are properties like **description** and **subjectOf**. Each of the eight base types has supplementary properties that are characteristic for the type and help interlinking it to other nodes. These are properties like author for **CreativeWork** or **birthDate** for **Person**. Some of the eight base types are further extended into subtypes. **MusicGroup** is an extension of **Organization**, for it has useful extra properties like album and genre. **MediaObject** is an extension of **CreativeWork** for properties like **contentUrl** or **productionCompany**. As **MediaObject** will probably be the most used type within the TROMPA project, is further extended with types like **AudioObject** and **MusicVideoObject**. Where the schema.org Types and properties are primarily for expressing meaningful categories and inter-relations in the CE, an **additional layer of metadata** properties are added to provide text fields that allow searches across the TROMPA data set. Adopted from the [Dublin Core Metadata Initiative](#) are the [15 classic metadata terms](#) (DCMES), each added to a selection of CE internal types as properties accepting String values. These properties are the primary metadata fields used for semantic searches in the CE: From the [Simple Knowledge Organization System](#) (SKOS) ontology, five of the [six mapping properties](#) are adopted to allow equivalence interlinking. In conjunction, the metadata property [language](#) and the schema.org property [inLanguage](#) are used to handle all current internationalization needs. To track the origin of a resource through its derivative(s), the nine base properties of the [PROV Ontology](#) were adopted on the relevant types: The [schema.org Thing](#) type has a standard [additionalProperty](#) property, which allows the extension of any type of node with any number of custom properties. This allows CE users to express their own data model in the CE, while ensuring the entered data adheres to CE metadata and interlinking standards. The type property of any thus related node allows to maintain RDF compatibility for these additional properties.

Section 3 describes the requirements for the data stored in the CE and it provides practical guidelines on managing data and the metadata model. The primary design concept to keep in mind is that the CE does not contain direct representations of real-life things, abstract entities or files. For

example, a **Person** type node in the CE does not represent a person directly, but represents a web resource that contains information about this person, and only about this person. It is not a problem that the same person, or the same audio file, is represented in the CE by multiple nodes; this only means that there are multiple web resources about this entity that are relevant to TROMPA. A composer might have several dedicated web pages in different public repositories, which all happen to be relevant web resources to TROMPA. Any data produced by TROMPA participants or users on the basis of this data will also be stored in the CE. Yet this data also only consists of references to web resources. In general, each node stored in the CE has to be a reference to a web resource. The value of any node property can be scalar (string, number, date etc.) or it can consist of one or several other nodes. Schema.org provides a generic model to interlink the base type entities with a variety of properties. Between two nodes, a number of relations can exist, each expressing a different or overlapping semantic relation to the other. It is important to create all such semantic relations between nodes, as this way it is possible to find the nodes through different types of (semantic) search queries. While the schema.org types and properties are mainly to provide semantic structure to the data in the CE, the **metadata properties** are there to provide for global searches on the basis of search terms. For this reason, all metadata properties accept scalar type values only. As a general rule, the metadata properties should in the first place contain information on the thing the web resource is about. All metadata properties should be written in the same language, and the *language* property should be set accordingly. **Interlinking between nodes** stored in the CE can and should be done through the default properties provided by the schema.org base types and the extension selected for the TROMPA project.

When importing a node for which there are already multiple equivalent nodes present, it is necessary to create bidirectional equivalence relations with all those equivalent nodes. From the [Simple Knowledge Organization System](#) we adopt the mapping ontology that allows to define several levels of equivalence. Within the TROMPA project, currently **six languages** need to be supported. Each node will have the metadata property *language*, which should be one of the six languages. When importing data in multiple languages, it is important to create proper **exactMatch** relations. In order to allow **ordered** and **unordered** lists (or collections) of items to be maintained in the CE database, the **ItemList** and **ListItem** types are supported. Regarding **provenance**, the CE internal model supports base level implementation of the PROV-O mechanism to track provenance for data stored in CE. Between the properties originating from the different ontologies, there are some apparent **redundancies**. For now, we prefer to leave all these redundancies in the model, because it is not yet clear which properties are always to get the same value. In time, we will choose whether to remove completely redundant properties in favour of one. All nodes and relations used in the CE internal data model can be **traced back** to a well-known ontology and have a RDF URI. The aim is to maintain this RDF compatibility throughout the TROMPA project. In order to ensure **consistency** in the data added to the CE by each partner, we **provide concrete guidelines** about what fields to set on the object types which we are currently using. Regarding **data privacy**, the data stored within the CE's graph database is assumed to be public in nature, with the creation and interlinking of open data forming a core focus of the TROMPA project. Nevertheless, certain user data pertaining to TROMPA will need to remain private, or accessible to only particular specified users. Examples include private rehearsal recordings that instrumental players or choir singers may wish to listen to in order to support rehearsal practice. Such requirements will be supported by mandating storage of non-public data in web-accessible locations outside of the CE, tied into the CE only by reference (URI). Fine-grained user-based access authorization can then be implemented at the external stores.

Candidate technologies for implementation of the authorised external storage include Solid PODS (“personal online data stores”), and S3 buckets implemented on the Amazon AWS Cognito platform.

Section 4 describes the GraphQL interface of the CE for managing the data. [GraphQL](#) 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. GraphQL supports three types of functionalities that are accessible through the GraphQL API interface, namely **queries**, **mutations** and **subscriptions**. For the CE API, a [graphic interface](#) is available that provides a human-friendly way to interact with the GraphQL API interface and supports rich introspection of the schema. Moreover it offers an overview for all the ‘custom’ functionalities available, like adding, mutating or deleting specific node relations. **Queries** are requests for existing data from the database and can consist of the Type of entity for which is queried, the conditions and a list of properties to be included in the response. **Mutations** are queries that add, update, remove data in the database, or adding / removing a relation between nodes. **Subscriptions** are used to run specific algorithms from WP3 items that exist in the CE. The current version (0.4) of the CE-API GraphQL interface does not yet support authentication. This means that all data can be accessed, changed and added by anybody. When the (hosted) CE-API is made available to the general public, access control and authentication will be added. Detailed examples for **queries**, **mutations** and **subscriptions** are provided in this deliverable.

As described in section 5 the CE api provides a very basic **REST interface**. The main purpose of this REST interface is to provide a unique URL for each node in the CE database and to provide JSON-LD output. Adjacent nodes in the graph which are related to the one requested will be included in the response only by reference to their respective REST URL.

Section 6 is dedicated to the integration of jobs and processes in the CE. As described in [D5.1 Data Infrastructure](#) and [D5.3 TROMPA Processing Library](#), Music Information Retrieval (MIR) technologies as developed in WP3, and Crowd-powered improvements as developed in WP4 are ultimately to be integrated with the CE. The CE data model in combination with the CE GraphQL interface enables component and ultimately (pilot) application developers to create nodes in the CE database that could serve as jobs for WP3 and WP4 technologies to be picked up and processed. In turn, WP3 and WP4 developers can set up a system to retrieve those jobs, to be executed against data referenced in the CE. After completion of the job, references to the results can be written back as nodes in the CE database. Subsequently, relations can be created between those results and the larger TROMPA data set. A scalable and generic solution is created for Component-CE-WP3/4 integration. This solution, as presented in this chapter, provides a standardised method for task/job creation and retrieval and allows both components and WP3/4 systems to handle jobs in real time or in batches in asynchronous fashion. This generic solution comprises of the following steps:

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

A [subscription](#) mechanism, can enable both Component and Process system to be actively updated on job creation and status updates in real time. This way, the CE becomes the intermediary of

Component-WP3/4 interactions. The data model supporting this approach is based on a [schema.org](#) compatible data model that can be broken down into three parts: a) **Public nodes** representing the data and the results of processes (e.g. data object Y), b) **Template nodes** which are maintained by WP3/4 developers and correspond to specific algorithms (e.g. algorithm X) and c) **Instance nodes** - created by Component(s) corresponding to specific tasks (e.g. run algorithm X to data Y). Details on these three types of nodes are given in subsection 6.2.

For **algorithm perspective**, the main responsibility would be to enter the correct template nodes into the CE database. With this set up, the algorithm process application can now detect whether a job is requested by querying regularly the CE database for new instances of template nodes (**ControlActions**). After a new request came in, the algorithm process application can then retrieve the necessary parameters and file(s) to act on and start writing back status or error updates on the **ControlAction** node that represents the job request. Regarding **components**, Component developers can query the CE database for **EntryPoints** that could potentially be interesting for its users. By implementing a user interface on the basis of the information in the (dynamic) template nodes **Property** and **PropertyValueSpecification**, the algorithm process (WP3/4) would become available for a user. **The role of the CE** in this mechanism is to maintain the data model and custom mutations that will enable Component and process algorithm application developers to create and follow **ControlActions** that effectively behave like jobs. This model should allow Component-WP3/4 interactions to take place as frictionless as possible, yet assuring the CE retains the position of middleman for all these interactions.

One of the major technical requirements of the project is that generic components can be reused in different pilots and end-user applications. At this point, we distinguish the following frontend (as in browser application) components that could and should be reused in the first release of pilot applications in M24.

- ❖ **CE Multimodal component.** This is a React library that can be re-used in a React Javascript project to have easy access to common search queries to the CE and visualisation of results of objects stored in the CE.
- ❖ **CE Digital Score Edition component.** This is a React library that can be used to render MEI scores as SVGs within a web client, and supports the creation and viewing of annotations upon the score.

Other components might be defined during the project. We agreed that the frontend components or libraries that could be of use to TROMPA partners should comply with the following technical requirements:

- ❖ The React component can be used with the latest version of React.
- ❖ The URL of the GraphQL endpoint targeted by the CE-API should be configurable without the need to compile from source.
- ❖ The React component will accept props to control the behavior of the component.
- ❖ The React component can be used as a Controlled Component
- ❖ The React component can be styled using a ThemeProvider which supports overwrites using JSS.

Version Log		
#	Date	Description
v0.1	16 Oktober 2019	Initial version
v0.2	20 Oktober 2019	Initial version submitted for internal review
v0.3	30 Oktober 2019	Revised version after internal review
v1.0	31 Oktober 2019	Final version submitted to EU

# Table of Contents

<b>Table of Contents</b>	<b>7</b>
<b>1. Introduction</b>	<b>9</b>
1.1 Naming conventions	9
1.2 Style conventions	9
<b>2. Internal data model</b>	<b>11</b>
2.1 Types and properties	11
2.1.1 Schema.org types and properties	11
2.1.2 Dublin Core metadata properties	13
2.1.3 SKOS equivalence linking properties	14
2.1.4 Internationalization properties	14
2.1.5 PROV-O provenance properties	14
2.1.6 Additional properties	15
<b>3. Data integration requirements and guidelines</b>	<b>15</b>
3.1 Type guidelines	15
3.2. Property guidelines	16
3.2.1 Base model properties	17
3.2.2. Core metadata properties	19
3.2.3. Equivalence linking properties	25
3.2.4 Internationalization properties[not yet implemented]	26
3.2.5. Item List	27
3.2.6. Provenance properties	28
3.2.7. Additional properties	30
3.3 Property redundancies guidelines	30
3.4 RDF compatibility guidelines	34
3.5 Ontological guidelines	34
Composition	34
3.6 Private data guidelines [not yet implemented]	36
<b>4. GraphQL interface for managing data</b>	<b>38</b>
4.1 Schema introspection	39
4.2. Queries	40
4.2.1 Simple query for one node	40
4.2.2 Simple query for multiple nodes	41
4.2.3 Complex query	42
4.3. Mutations	43
4.3.1 Creating a node	43

4.3.2 Updating a node	44
4.3.4 Deleting a node	44
4.3.5 Add a relation between nodes (primitive types)	45
4.3.6 Add a relation between nodes (Interfaced or Unioned types)	45
4.3.7 Remove a relation between nodes	46
4.4. Subscriptions [not yet implemented]	47
4.5 Authentication [not yet implemented]	47
<b>5. REST interface</b>	<b>47</b>
<b>6. Integration of jobs and processes</b>	<b>48</b>
6.1 Generic solution overview	48
6.2 Data model	50
6.2.1 Template nodes	50
6.2.2 Instance nodes	51
6.2.3 Public nodes	52
6.2.4 End result	52
6.3 Perspective of algorithm process application	54
6.3.1 GraphQL queries	54
6.3.1.1 Create and maintain template nodes	54
6.3.1.2 Monitor and update instance nodes	59
6.3.1.3 Complete the request response cycle	60
6.4 Perspective of Component	62
6.4.1 GraphQL queries	62
6.3.1.1 Query for available algorithm processes	62
6.3.1.2 Monitor instance nodes	63
6.5 Perspective of CE	67
<b>7. Integration of frontend components</b>	<b>67</b>
<b>8. Conclusion</b>	<b>68</b>
<b>9. References</b>	<b>68</b>
9.1 List of abbreviations	68

# 1. Introduction

This deliverable is the technical requirements and integration report and specifies the strategies for technical integration of data generated by different technologies. The initial version will include the integration guidelines to be followed during the project that will be updated in the final version.

The document is created by merging two earlier working documents (CE API Manual and CE Import Guidelines). The document is written for a technical audience and should provide practical information and guidelines for developers and researchers on how to integrate with the Contributor Environment (CE). It is expected that the document is updated and improved after its initial submission. The goal would be that at the end of the project, third parties with little background information on the project understand how they can integrate with the CE based on this technical documentation. A large part of this technical documentation is expected to be part of the M30 version of the public available deliverable ‘D5.1 Data infrastructure’.

The rest of this section provides naming and style conventions for this document. The main contents of the document start (section 2) with a detailed overview of the internal data model of the TROMPA Contributor Environment (CE). In section 3 it describes best practices for setting properties and relations when managing data in the CE in the form of guidelines. In sections 4 and 5 the interfaces for interacting with the CE are documented. The CE consists of an API application that exposes basic functionalities to update and query a graph database (Neo4j) that contains a dataset complying with the CE internal data model, which is based on the [schema.org](#) structured data vocabulary. These functionalities can be accessed through a GraphQL and a RESTful API interface. In chapter 6 it is explained how the job workflows and processes that are developed in WP3 and WP4 can be integrated with the CE. Chapter 7 provides requirements for the frontend components that can be reused in different end-user pilots.

## 1.1 Naming conventions

Following graph parlance, throughout this document some concepts are used which are also known under other names. Type is another word for Class. An instance of a Type (a record in the CE) will be called a node. Types have properties, which are like fields, or ‘columns’ in SQL terms. Properties are restricted to contain values only of one or several predefined Types. There are predefined scalar Types, like String, Boolean, Date or Number. A property can also be restricted to contain nodes of a certain Type. A node property that contains another node is in effect a relation to that other node, which is also known as an edge. When node X is connected to node Y by such a relation, node Y is considered to be adjacent to node X.

A property can contain one or multiple values. For scalar values, this would be an array. For nodes, this would be one or more relations to other nodes.

## 1.2 Style conventions

For clarity, in the following sections types and properties will be marked with the following styles:

**Type** (Bold, UpperCamelCase)

*property*      (Italic, camelCase)

## 2. Internal data model

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. 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.

### 2.1 Types and properties

#### 2.1.1 Schema.org types and properties

The CE internal data model is primarily based on the schema.org. The Types and properties adopted from schema.org are the foundation for expressing TROMPA relevant web resources as clearly defined entities, and for interlinking those entities in a meaningful way.

All nodes in the CE will be either one of the eight base types from schema.org.

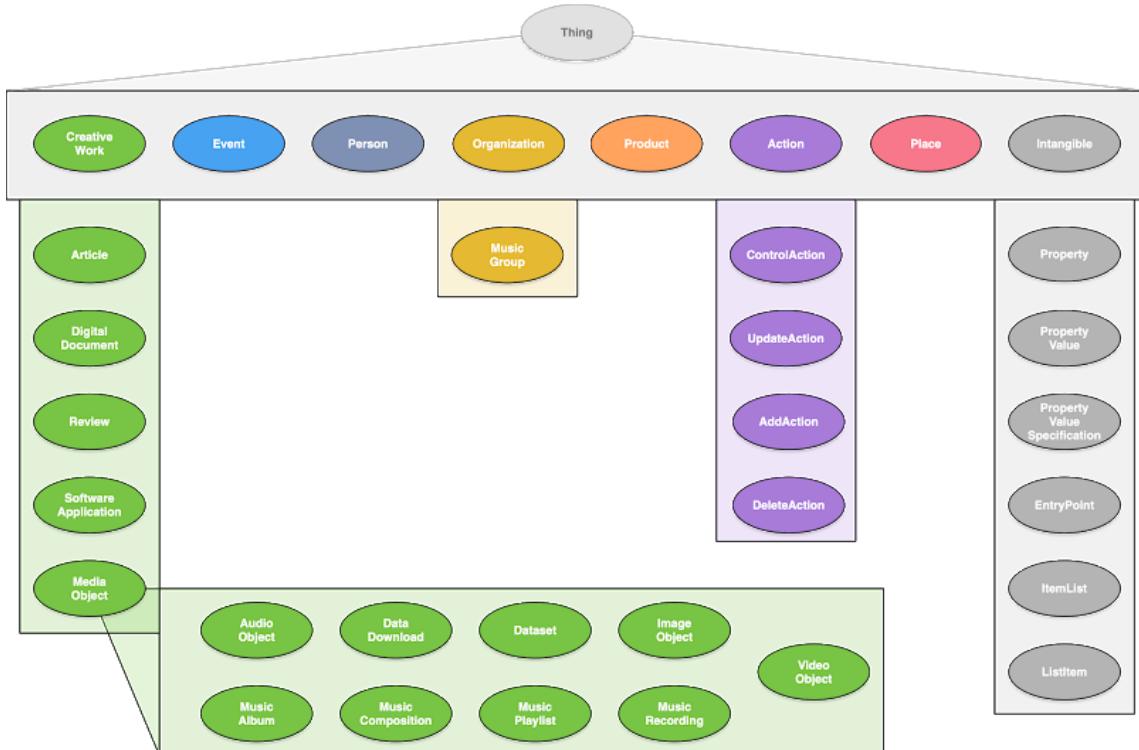
These eight types are extended from schema.org's ultimate base type [Thing](#). **Thing** is not available as a type to create a node from in the CE.

**Thing** has a number of properties which are available for all types, and thus all nodes that reside in the CE. These are properties like *description* and *subjectOf*.

Each of the eight base types has supplementary properties that are characteristic for the type and help interlinking it to other nodes. These are properties like *author* for **CreativeWork** or *birthDate* for **Person**.

Some of the eight base types are further extended into subtypes. **MusicGroup** is an extension of **Organization**, for it has useful extra properties like *album* and *genre*. **MediaObject** is an extension of **CreativeWork** for properties like *contentUrl* or *productionCompany*.

As **MediaObject** will probably be the most used type within the TROMPA project, is further extended with types like **AudioObject** and **MusicVideoObject**.



**Figure 2.1.** Schema.org base types, plus TROMPA relevant extensions

The following types will be supported in the CE. Please, follow the links to schema.org for an overview of the properties and value types available for each type.

- ❖ **Action**
  - [ControlAction](#)
  - [AddAction](#)
  - [ReplaceAction](#)
  - [DeleteAction](#)
- ❖ **CreativeWork**
  - [Article](#)
  - [Dataset](#)
  - [DigitalDocument](#)
  - **MediaObject**
    - [AudioObject](#)
    - [DataDownload](#)
    - [Dataset](#)
    - [ImageObject](#)
    - [MusicAlbum](#)
    - [MusicComposition](#)
    - [MusicPlaylist](#)

- [MusicRecording](#)
- [VideoObject](#)
- [SoftwareApplication](#)
- [VideoObject](#)
- ❖ [Event](#)
- ❖ [Intangible](#)
  - [Property](#)
  - [PropertyValue](#)
  - [PropertyValueSpecification](#)
  - [EntryPoint](#)
  - [ItemList](#)
  - [ListItem](#)
- ❖ [Organization](#)
  - [MusicGroup](#)
- ❖ [Person](#)
- ❖ [Place](#)
- ❖ [Product](#)

### 2.1.2 Dublin Core metadata properties

Where the schema.org Types and properties are primarily for expressing meaningful categories and inter-relations in the CE, an additional layer of metadata properties are added to provide text fields that allow searches across the TROMPA data set.

Adopted from the [Dublin Core Metadata Initiative](#) are the [15 classic metadata terms](#) (DCMES), each added to a selection of CE internal types as properties accepting String values. These properties are the primary metadata fields used for semantic searches in the CE:

- ❖ [\*title\*](#)
- ❖ [\*creator\*](#)
- ❖ [\*subject\*](#)
- ❖ [\*description\*](#)
- ❖ [\*publisher\*](#)
- ❖ [\*contributor\*](#)
- ❖ [\*date\*](#)
- ❖ [\*type\*](#)
- ❖ [\*format\*](#)
- ❖ [\*identifier\*](#)
- ❖ [\*source\*](#)
- ❖ [\*language\*](#)
- ❖ [\*relation\*](#)
- ❖ [\*coverage\*](#)
- ❖ [\*rights\*](#)

### 2.1.3 SKOS equivalence linking properties

From the [Simple Knowledge Organization System \(SKOS\)](#) ontology, five of the six [mapping properties](#) are adopted to allow equivalence interlinking.

The CE will contain mainly web references and does not strive to be an authoritative (new) public source of ground truth. Practically, this means that for a given composer, say ‘Gustav Mahler’, there will not be one node that represents the person Mahler. There will be multiple nodes for Mahler, each one representing a web resource; There will be one node representing the WikiData page about Mahler, another representing the MusicBrainz page about Mahler in English, and yet another representing the MusicBrainz page in French.

The SKOS mapping properties provide for a way to relate nodes (web resources) that refer to the same thing or abstract entity. From a user or search perspective, these mapping relations will allow to consider nodes that are interrelated through these equivalence relationships, as one and the same. It will allow to show for example all Mahler compositions, regardless of whether they are related to the WikiData or to the MusicBrainz page of Mahler.

For each type, the ...Match properties accept only the same type as its host node. The relatedMatch property accepts any node type.

- ❖ [exactMatch](#)
- ❖ [closeMatch](#)
- ❖ [broadMatch](#)
- ❖ [narrowMatch](#)
- ❖ [relatedMatch](#)

### 2.1.4 Internationalization properties

In conjunction, the metadata property [language](#) and the schema.org property [inLanguage](#) are used to handle all current internationalization needs.

### 2.1.5 PROV-O provenance properties

To track the origin of a resource through its derivative(s), the nine base properties of the [PROV Ontology](#) were adopted on the relevant types:

- ❖ [wasGeneratedBy](#)
  - All types and extensions
- ❖ [wasDerivedFrom](#)
  - All types and extensions
- ❖ [wasAttributedTo](#)
  - All types and extensions
- ❖ [Used](#)
  - All types and extensions
- ❖ [wasAssociatedWith](#)
  - **Person**
  - **Organization** and extension
- ❖ [actedOnBehalfOf](#)
  - **Person**
  - **Organization** and extension

- ❖ *startedAtTime*
  - Action
- ❖ *wasInformedBy*
  - Action
- ❖ *endedAtTime*
  - Action

## 2.1.6 Additional properties

The schema.org **Thing** type has a standard *additionalProperty* property. We use this property to add any number of **PropertyValue** type nodes. This type accepts a *propertyID* property that should contain the name of the additional property, while the *value* property contains either a scalar or another node. The *type* property of this additional property node can contain any RDF URI.

This additional property mechanism allows the extension of any type of node with any number of custom properties. This allows CE users to express their own data model in the CE, while ensuring the entered data adheres to CE metadata and interlinking standards. The *type* property of any thus related node allows to maintain RDF compatibility for these additional properties.

### 3. Data integration requirements and guidelines

This chapter describes the requirements for the data stored in the CE and it provides practical guidelines on managing data and the metadata model.

#### 3.1 Type guidelines

When handling CE data, the primary design concept to keep in mind is that the CE does not contain direct representations of real-life things, abstract entities or files. The data contained in the CE represents web resources. In their turn, these web resources can represent real-life things or abstract entities, or be files.

Thus, a **Person** type node in the CE does not represent a person directly, but represents a web resource that contains information about this person, and only about this person. Similarly, an **AudioRecording** type node does not contain audio file data, but is a reference to an audio file that is available at a public URL. This is not to say that a **Person** type node in the CE only contains a URL and not the person's birth date, or that an **AudioRecording** type node can not contain the title of the performance from which the audio was recorded. But this data is merely metadata about the web resource or metadata scraped from the web resource itself. This metadata allows relevant searches and interlinkage of the web resource references contained in the CE.

It is not a problem that the same person, or the same audio file, is represented in the CE by multiple nodes; this only means that there are multiple web resources about this entity that are relevant to TROMPA. These web resources might have complementary information about the same entity, or we want to compare the web resources.

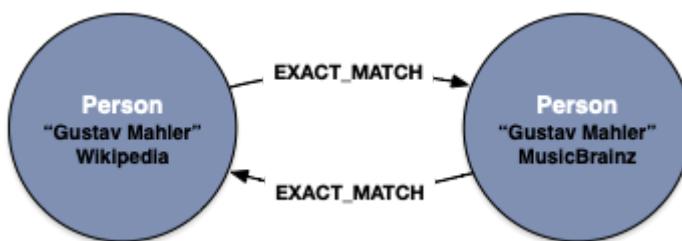


Figure 3.1

Another way to look at this, is to consider the data in the CE as a mapping of musical data that is available on the web, and which happens to be relevant to TROMPA participants or users.

A composer might have several dedicated web pages in different public repositories, which all happen to be relevant web resources to TROMPA. Each of these web resources would then be represented by a **Person** type node in the CE, while none of these nodes represents the composer directly, or would be the main node for that composer. Through interlinking, these nodes can be marked as being web resources about the exact same entity. When querying the CE for this composer, all these web resources will come up in equivalent fashion.

Any data produced by TROMPA participants or users on the basis of this data will also be stored in the CE. Yet this data also only consists of references to web resources;

An alignment file produced by a participant will be stored at a public URL and can be accessed through the CE as a **CreativeWork** type node, which only contains this URL and some metadata.

An annotation created by a TROMPA user will be an **Annotation** type node containing the public RESTful URL to itself in the CE, which exposes the annotation content, be it flat text or a reference to an uploaded file.

In general, each node stored in the CE has to be a reference to a web resource. Any given web resource can only have one node in the CE. The URL to this web resource is a required property (source) when importing nodes of any type, and is validated to be unique. It is not possible to store multiple nodes with the same resource URL in the CE.

The subsequent chapters will describe best practices of how to apply this design principle consequently from the perspective of the various types of data that can be stored along with references and from the perspective of interlinking these references.

## 3.2. Property guidelines

The value of any node property can be scalar (string, number, date etc.) or it can consist of one or several other nodes. In a graph database, a scalar property is expressed as a property value that resides inside a node, like a *name* or a *birthdate*. If a property value consists of another node, this is expressed as a ‘relation’ with a label derived from the property name (caps snake case). Though relations can be bidirectional, currently only unidirectional relations are supported.

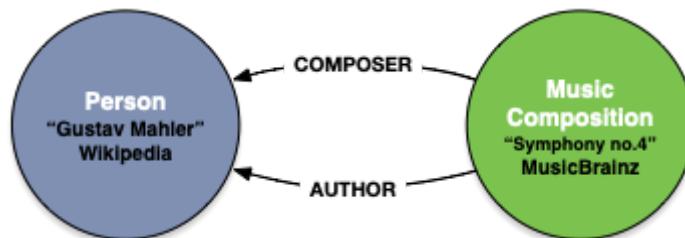


Figure 3.2

When a property value consists of multiple nodes, this is expressed as multiple relations with the same property name.

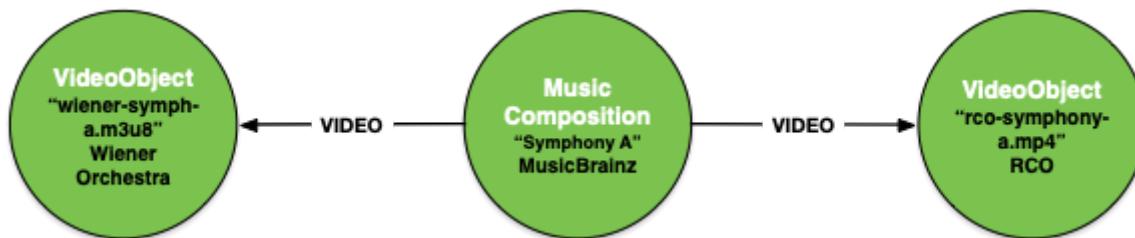


Figure 3.3

Currently, relations themselves do not contain properties. This will be supported later for TROMPA specific use cases like segmentation and curation. Relation properties can only be Scalar values. The aim is to avoid setting relation properties when importing data.

### 3.2.1 Base model properties

Schema.org provides a generic model to interlink the base type entities with a variety of properties. Between two nodes, a number of relations can exist, each expressing a different or overlapping semantic relation to the other. It is important to create all such semantic relations between nodes, as this way it is possible to find the nodes through different types of (semantic) search queries.

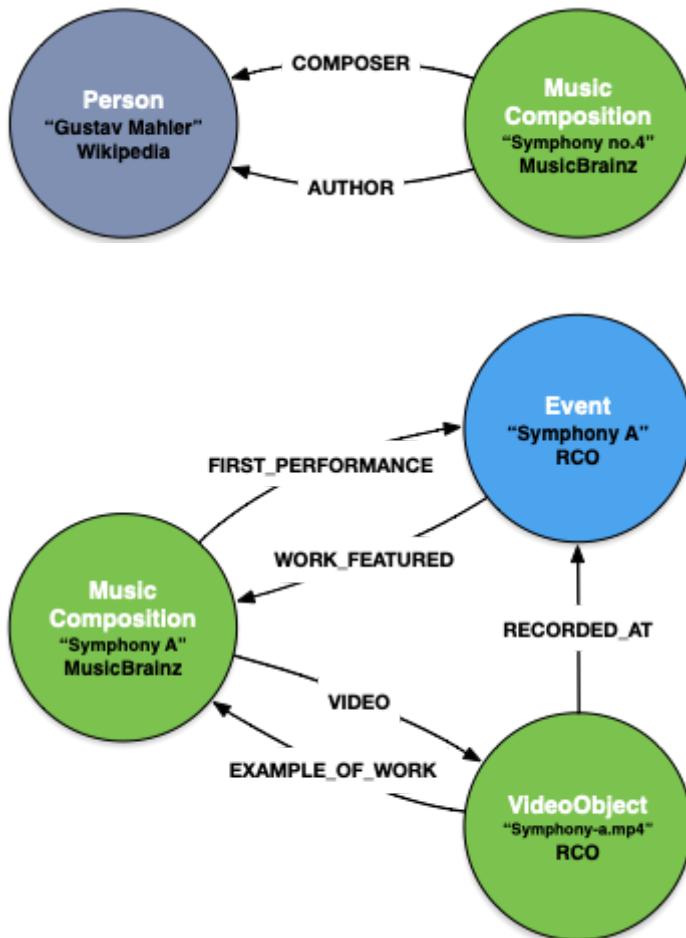


Figure 3.4

Nodes represent web resources, and a specific thing could have several web-resources dedicated to it. The exactMatch property is used to create relations between nodes that are about the same thing. When linking a node to one of those nodes, it is not necessary to duplicate the relations to the exact matching nodes. Creating relations with just one of the exact matching nodes is sufficient.

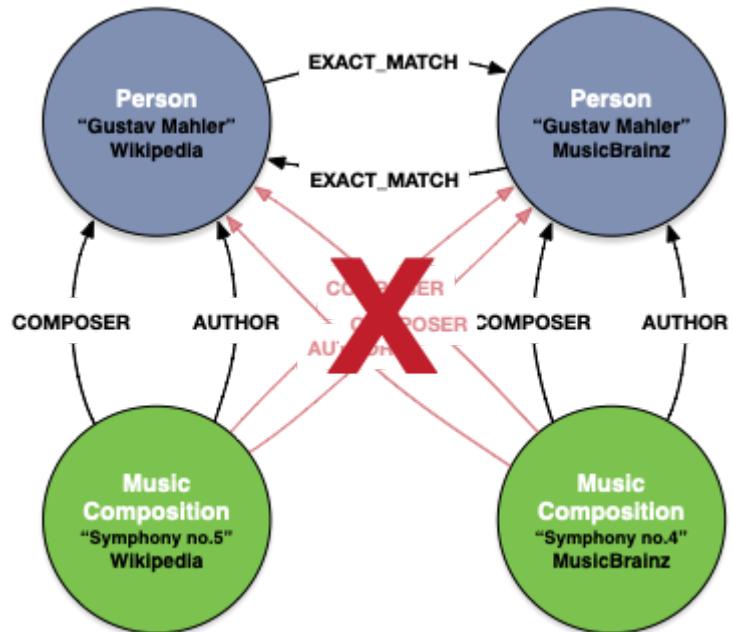


Figure 3.5

### 3.2.2. Core metadata properties

While the schema.org types and properties are mainly to provide semantic structure to the data in the CE, the metadata properties are there to provide for global searches on the basis of search terms. For this reason, all metadata properties accept scalar type values only.

Any node in the CE represents a web resource. A node's metadata properties contain information about either the web resource itself or about a thing (real or abstract) or the file that this web resource is about.

As a general rule, the metadata properties should in the first place contain information on the thing the web resource is about. If this does not make sense (e.g. who would be the *creator* of a **Person**, like a composer?) the metadata property should contain information about the web resource (So the *creator* of a Wikipedia page about a composer would be Wikipedia).

Some metadata properties never contain information on the thing the web resource is about, like *contributor* (this is always the agent that created, or is the current maintainer of, the web resource) and *source* (this is always the URL of the web resource).

All metadata properties should be written in the same language, and the *language* property should be set accordingly, with the corresponding 2-letter language code. This practice ensures that searches can be done within the context of a single language.

As the proper setting of these properties is essential for a well functioning CE, there is a guideline section for each of the metadata properties:

Property	Type	On	Example(s)	Explanation
<i>title</i>	String	thing	'Symfony No. 2'	A name given to the thing the web resource is about. Typically, a Title will be a name by which the resource is formally known.
			'Symfony No. 2 for 2 voices, mixed chorus, orchestra, for voices ...'	The title should not be too long. The title will be displayed at the top of a search result, with (part of) the description.
			'Piano recording Für Elise'	When the web resource is an uploaded file, the title should be about the content of the file. Preferably let the uploader define this title.
<i>creator</i>	String	thing	'Gustav Mahler'	The person, organization or service who created the thing the web resource is about. The name of entity primarily responsible for creating the 'thing' the web resource is about. This name should be entered in givenName-familyName format. If the creator of the thing is ambiguous (e.g. who would be the creator of

				'Gustav Mahler') then enter the creator of the web resource or the service. This name should be entered as the base URL for the web resource, or the service.
			'Mahler, Gustav'	The creator name should follow givenName-familyName order
			'https://www.voctrolabs.com'	When the web resource is for example a file auto-generated by a TROMPA partner service, identify the creator as the participant, identified by the base URL.
			'https://en.wikipedia.org'	For a web resource about a Person, for example 'Gustav Mahler', the public repository itself can be identified as the creator, by its base URL
			'https://api.trompamusic.eu/b670c593-d1c6-45ed-9a09-6a51c12108e1'	If the creator is represented by a node in the CE, for example a TROMPA user, this node's URL can be used to identify the creator
<i>subject</i>	String	thing	'symphonic poem,Gustav Mahler,Berlin Philharmonic'	The topic of the resource. Typically, the subject will be represented using keywords, key phrases, or classification codes. Recommended best practice is to use a controlled vocabulary.
			'https://en.wikipedia.org/about_gustav'	An URL is not a valid keyword
			'This is about a composer born in 1867 in Linz...'	Stick to keywords only. The <i>description</i> property allows a flowing test.
<i>description</i>	String	thing	'Mahler completed what would become the first movement of the symphony in 1888 as a single-movement symphonic poem...'	An account of the resource. Description may include but is not limited to: an abstract, a table of contents, a graphical representation, or a free-text account of the resource.
			'symphonic poem,Gustav Mahler,Berlin Philharmonic'	This should be free flowing text. For keywords use the <i>subject</i> property

<i>publisher</i>	String	thing	'Friedrich Hofmeister, Leipzig'	<p>The person, organization or service responsible for making available the thing the web resource is about.</p> <p>Typically, the name of the Publisher should be used, if possible with a place indication.</p> <p>If the publisher of the thing is ambiguous (e.g. who would be the publisher of 'Gustav Mahler') then enter the publisher of the web resource or the service. This name should be entered as the base URL for the web resource, or the service. In this case, the creator, publisher and contributor are often the same.</p>
			'https://en.wikipedia.org'	If the web resource is for example a page about a composer, a publisher for the composer does not make sense. Mark the web resource base url as the publisher.
			'https://imslp.org'	If the web resource is about, for example, a pdf from a score published in paper, mark the paper's publisher, not the base URL of the website where the PDF can be found.
			'https://musescore.org'	If the web resource is, for example, a digitized score published on a website, mark the base URL of the website.
<i>contributor</i>	URL	resource	'https://imslp.org'	<p>A person, an organization, or a service responsible for contributing the thing to the web resource. This can be either a name or a base URL.</p> <p>If the contributor of the thing is ambiguous (e.g. who would be the contributor of 'Gustav Mahler') then enter the contributor to the web resource about the thing, or the entity using the service to create the thing. This should be a URL unambiguously pointing to the contributor.</p> <p>If this contributor to the web resource or service is unknown, enter the web resource or the service itself as contributor. This name should be entered as the base URL for the web resource or service. In this case, the</p>

				creator, publisher and contributor are often the same.
			'https://imslp.org/wiki/List_of_works_by_Gustav_Mahler'	The contributor should be only the base URL.
			'IMSLP'	Enter the full base URL
			'https://api.trompamusic.eu/b670c593-d1c6-45ed-9a09-6a51c12108e1'	The public profile of a TROMPA user is a valid contributor identifier
<i>date</i>	Date	thing	'1895-12-13'	A point in time associated with an event in the lifecycle of the resource. Must be in 'YYYY', 'YYYY-MM' or 'YYYY-MM-DD' format. Examples: composition first performance, publishing date, birth date, release date, file generation date, annotation date
			'1895'	Is also a valid date
			'2018-12-04 12:04:11'	Is not a valid Date
<i>type</i>	URL	thing	'http://purl.org/ontology/mo/Composition'	The RDF type URI of the node. Note: this will be a secondary type, as the primary type will correspond to the CE internal model type from schema.org and is set automatically. Additional types can be set in the <i>additionalType</i> property.
			'mo:composition'	Turtle notation is not supported
			"	Can be left empty
<i>format</i>	String	thing	'audio/aac'	An Internet Media Type [MIME]
			'text/html'	If the thing the web resource is about has a mime-type, enter this mime-type. If the web resource is, for example, a wikipedia page about a composer, the mime type for the composer does not make sense. Mark the mime type of the web page.

			'1140x300 pixels'	Should be a valid mime type ( <a href="https://www.iana.org/assignments/media-types/media-types.xhtml">https://www.iana.org/assignments/media-types/media-types.xhtml</a> ) If necessary, we can define a custom mime type (vnd.trompamusic.[type])
<i>identifier</i>	UUID	CE	'5d05bfda-c050-424e-9d11-314b80225ea8'	An unambiguous reference to the resource within a given context. For CE we will use UUID. An invalid or non-unique UUID will fail validation. If no UUID is passed, one will be generated by the CE (recommended).
			' <a href="https://imslp.org/gustav_mahler">https://imslp.org/gustav_mahler</a> '	Even though unique, this is not a valid identifier
			'musicbrainz_adcdc472-8b19-4e6f-aa4e-be8c6aea5f8a'	This is not a valid UUID. One could imagine passing a Musicbrainz UUID, though. As long as this UUID is valid and unique for CE, it will work.
			"	When left empty, the CE will generate a UUID (recommended)
<i>source</i>	URL	resource	' <a href="https://imslp.org/wiki/Symphony_No.2_(Mahler,_Gustav)">https://imslp.org/wiki/Symphony_No.2_(Mahler,_Gustav)</a> '	The URL of the web resource to be represented by the node.
			' <a href="https://api.trompamusic.eu/b670c593-d1c6-45ed-9a09-6a51c12108e1">https://api.trompamusic.eu/b670c593-d1c6-45ed-9a09-6a51c12108e1</a> '	Any TROMPA produced data stored in CE will automatically have a unique URI made up of the TROMPA API base URL and the UUID of the node. Only use this for nodes that do not have any other (reliable & unique) URL to identify them with (like user annotations or automatically generated content).
			'IMSLP'	Enter the full unique base URL corresponding to the web resource
<i>language</i>	enum	metadata	'en'	The language the metadata is written in. This does not have to correspond to the language of the thing the websource is about. English metadata can be written about a music composition that has lyrics in German. In CE we use a fixed list with RFC4646 2-letter codes, currently: en,es,ca,nl,de,fr

			'english'	Is not a valid language code.
			"	Required value. Even if setting a language for the resource (eg a violin recording) does not make sense, the language indicates the language the metadata is written in, or the context from which the resource was created. (An uploaded recording from a Spanish interface would have 'es' as value)
			'uz'	Uzbek is not a supported language
<i>relation</i>	URL	resource	'https://imslp.org/wiki/Category:Mahler,_Gustav'	A related resource. In CE, any web resource can be used as a relation.
			"	Can be left empty
			'Gustav Mahler'	Should be a valid URL
<i>coverage</i>	String	resource	'world'	The spatial or temporal topic of the resource, the spatial applicability of the resource, or the jurisdiction under which the resource is relevant. Spatial topic and spatial applicability may be a named place or a location specified by its geographic coordinates. Temporal topic may be a named period, date, or date range. A jurisdiction may be a named administrative entity or a geographic place to which the resource applies. Recommended best practice is to use a controlled vocabulary such as the Thesaurus of Geographic Names [TGN]. Where appropriate, named places or time periods can be used in preference to numeric identifiers such as sets of coordinates or date ranges.
			'https://api.trompamusic.eu/c984a101-799f-422d-aec2-791c44e67dee'	A reference to a <b>Place</b> node in the CE can be used as a coverage area. Make sure the UUID points to a <b>PLACE</b> node.
<i>rights</i>	URL	thing	'https://en.wikipedia.org/wiki/Wikipedia:Text_of_Creative_Com	Information about rights held in and over the thing the web resource is about.

			mons_Attribution-Sha reAlike_3.0_Unporte d_License'	Typically, rights information includes a statement about various property rights associated with the resource, including intellectual property rights, and should be available as a document at the given URL. If rights for the thing are ambiguous (e.g. what are the rights for person 'Gustav Mahler')? then enter the rights for the web resource or the service.
			'Creative Commons'	Pass a full URL to the rights document
			"	Left empty means that no copyrights apply to this data. This is rare and we should strive to always have rights indicated.

**Table 3.1** List of core metadata properties and examples

### 3.2.3. Equivalence linking properties

Most interlinking between nodes stored in the CE can and should be done through the default properties provided by the schema.org base types and the extension selected for the TROMPA project ([Section 2.3.1](#)).

Although the default [\*sameAs\*](#) property could be abused for this purpose, schema.org does not provide for a way to assert equivalence between entities, let alone indicate more subtle types of 'same as' relations.

When importing a node for which there are already multiple equivalent nodes present, it is necessary to create bidirectional equivalence relations with all those equivalent nodes. This will chain the new node in equivalence with all those nodes. When future crowd curation determines a node does not fit in the equivalence chain, the equivalence relation(s) will be suppressed.

From the [Simple Knowledge Organization System](#) we adopt the mapping ontology that allows to define several levels of equivalence:

Property	Value type	Explanation/example
<i>exactMatch</i>	Parent type	'used to link two concepts, indicating a high degree of confidence that the concepts can be used interchangeably across a wide range of information retrieval applications.' <i>exactMatch</i> is a symmetric property; if set in a node, it should also be set in the related node. (until we manage to support bidirectional relations) If set, none of the other ... <i>Match</i> properties should be set.

		Two web resources about the same composer should each have the <i>exactMatch</i> property set with the other.
<i>closeMatch</i>	Parent type	<p>'used to link two concepts that are sufficiently similar that they can be used interchangeably in <b>some</b> information retrieval applications'</p> <p><i>closeMatch</i> is a symmetric property; if set in a node, it should also be set in the related node. (until we manage to support bidirectional relations)</p> <p>If set, none of the other ...<i>Match</i> properties should be set.</p> <p>Two score versions of the same music composition should both have the <i>closeMatch</i> property set with the other.</p>
<i>broadMatch</i> <i>narrowMatch</i>	Parent type	<p>'used to state a hierarchical mapping link between two concepts.'</p> <p><i>broadMatch</i> and <i>narrowMatch</i> are inverses; when setting either in one node, the inverse should be set in the related node.</p> <p>If either is set, none of the other ...<i>Match</i> properties should be set.</p> <p>A node for full score can have the <i>narrowMatch</i> property set with a node representing one page of the same score. The one page node could then set the <i>broadMatch</i> property with other.</p>
<i>relatedMatch</i>	<b>ThingInterface</b> (any base and extended type)	<p>'used to state an associative mapping link between two concepts.'</p> <p>If set, none of the other ...<i>Match</i> properties should be set.</p> <p><i>relatedMatch</i> is a symmetric property; if set in a node, it should also be set in the related node. (until we manage to support bidirectional relations)</p> <p>A music group that operates under different names and/or occupancies can have different nodes that are interrelated by <i>relatedMatch</i> relations.</p> <p>Should not be set when any of the other ...<i>Match</i> properties are set.</p>

**Table 3.2.** Equivalence and examples

### 3.2.4 Internationalization properties<sup>[not yet implemented]</sup>

Within the TROMPA project, currently six languages need to be supported. Each node will have the metadata property language, which should be one of the six languages. Setting the request Accept-Language header to one of those languages will yield language filtered results<sup>[not yet implemented]</sup>. Omitting the Accept-Language header will yield results for all languages.

When importing data in multiple languages, it is important to create proper exactMatch relations. If, for example, 3 WikiData pages for 'Gustav Mahler' are imported in 3 different languages as Person type nodes, these nodes should be interlinked by bidirectional exactMatch relations. Whether exactMatch relations were set or not, without Accept-Language header a query for the Person 'Gustav Mahler' would yield the WikiData pages for all 3 languages, and could include nodes related

to any of those 3 Person nodes. However, no exactMatch relations were set and the Accept-Language was set to 'fr', only the French version of the 'Gustav Mahler' page would be returned, with only French nodes related to the French version, and no nodes related to the 'Gustav Mahler' nodes in other languages. So, making sure the exactMatch relations are set correctly will ensure the 3 Person nodes are considered matching the same 'concept', and the results can include relevant nodes that are related to the non-French 'Gustav Mahler' nodes.

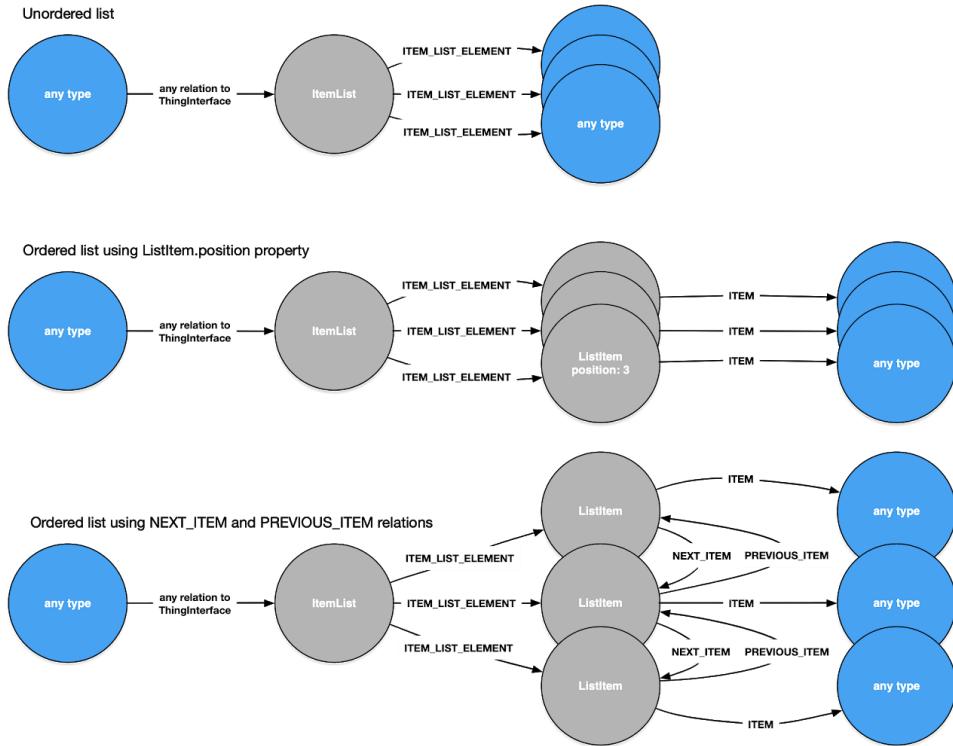
In the example above, the question might arise if, with a French Accept-Language header set, any content without French metadata is suppressed from the result. A relevant score file, for example, and for which language is irrelevant, might be suppressed because its metadata happens to be in German. For these cases, the inLanguage property is used on any CreativeWork and Event type nodes, plus their extended types. This inLanguage property can be set with any language, and can also be left empty. With the French Accept-Language set, the results would include nodes that have either of the language or inLanguage properties set to 'fr', OR the inLanguage property left empty. In the example above, the score with German metadata would have its inLanguage property left empty and it would be included in the French results.

The Accept-Language header can contain multiple language codes in order of priority. Results will be filtered for the first priority language<sup>[not yet implemented]</sup>. If a node is only available in the second language, this one will be returned, etcetera. Setting the Accept-Language to 'fr,en' would yield French results, and where French content is not available, default to English<sup>[not yet implemented]</sup>.

### 3.2.5. Item List

In order to allow ordered and unordered lists (or collections) of items to be maintained in the CE database, the **ItemList** and **ListItem** types are supported. There are several ways to represent a list of items in the CE database:

- ❖ Unordered lists can be created by adding an ItemList node, which then relates to all the list items through the *itemListElement* property.
- ❖ Ordered lists using the **ItemList.position** property. An arbitrary number of items can be added to the ItemList by creating a ListItem node in between each **Item** and the **ItemList** node. The **ListItem.item** property relates each **Item** to its respective **ListItem**. The order of the items can be determined through the **ListItem.position** property. This method is good for lists that remain static, as adding an item or changing the order is cumbersome.
- ❖ Ordered lists using the **ListItem.nextItem** and **previousItem** properties. This method is easier for dynamic lists, as it needs less updates to add, change or remove an item. Although technically it would only be necessary to create a single *itemListElement* relation between Itemlist and one ListItem, it is not advised: It would be difficult to use the CE API GraphQL interface to retrieve all items through concurrent *nextItem* relations.

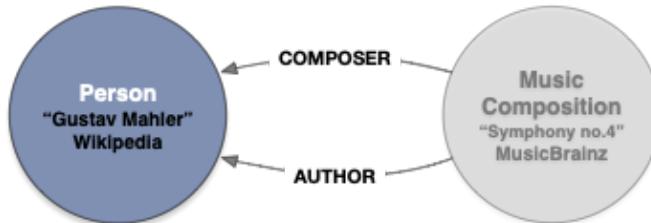


**Figure 3.6**

### 3.2.6. Provenance properties

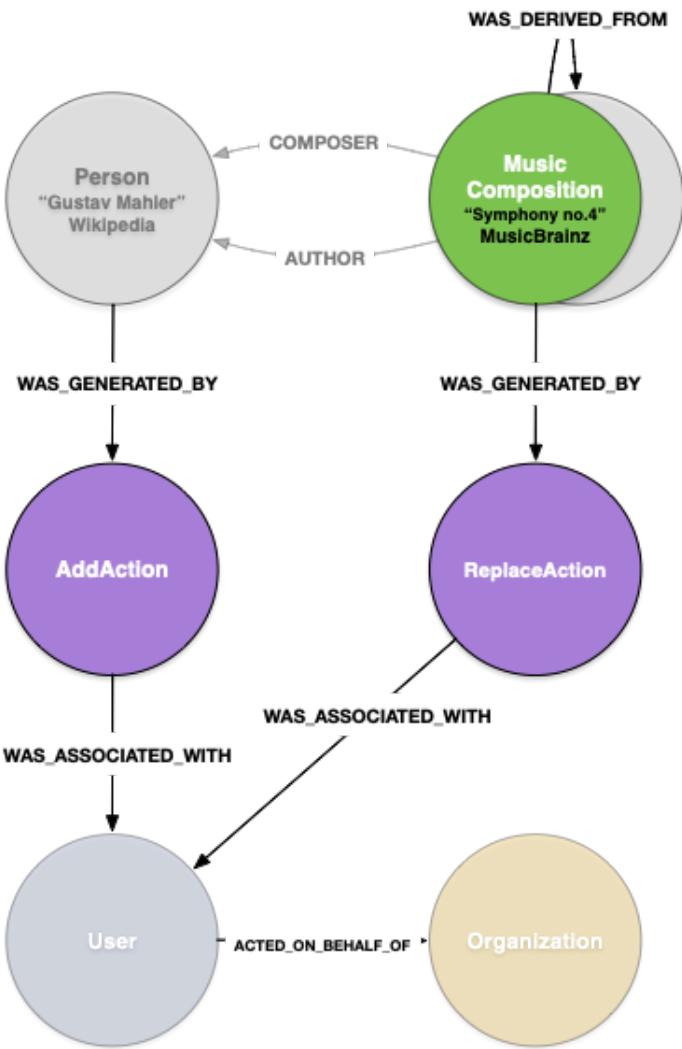
The CE internal model supports base level implementation of the PROV-O mechanism to track provenance for data stored in CE.

For most client usage, the CE will handle the creation of provenance tracking data itself<sup>[not yet implemented]</sup>, based on the user and the mutations this user applies to the dataset through the GraphQL interface. For example, when an existing **User** adds an additional **Person** node, connected to an existing **MusicComposition**:



**Figure 3.7**

The CE would make sure the following additional nodes and edges are also created:



**Figure 3.8**

For some CE clients, this functionality will not be sufficient. Data imported straight from a non-TROMPA public repository will have to be marked as originating from or related to **Agents** that cannot be derived from the creation call. The client might have to create a new **Agent**, set properties or add **Action** type data to set the proper basis for further provenance tracking.

As this is a rich subject, please read this PROV-O [primer](#), while keeping in mind that only the [Starting Point Terms](#) are implemented in the CE.

An example of custom added provenance data (**AddAction**):

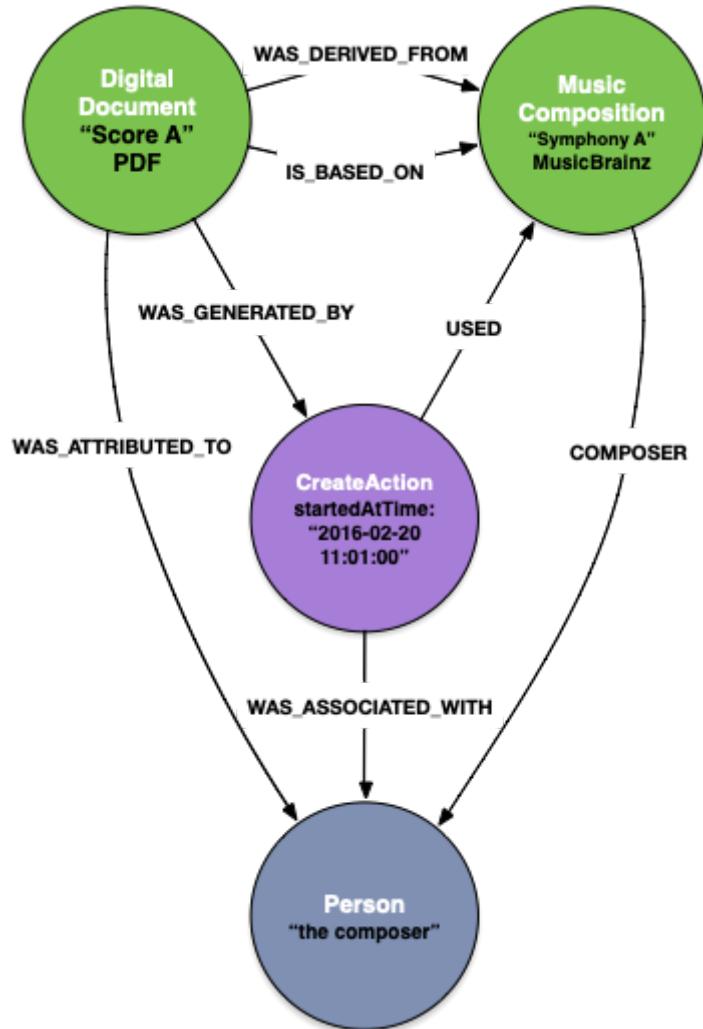


Figure 3.9

### 3.2.7. Additional properties

An additional property can be added to any single node by adding a node of type [PropertyValue](#) to the node's *additionalProperty* (ADDITIONAL\_PROPERTY relation). Use the **PropertyValue.propertyID** for the name of the value.

If the extra property is a relation to another node, create a relation to the target node with the **PropertyValue.nodeValue** property and set the **PropertyValue.valueReference** to the type of the targeted node.

If the extra property is a scalar, then set the **PropertyValue.value** to the stringified value, then set the property scalar type in **PropertyValue.valueReference**.

## 3.3 Property redundancies guidelines

Between the properties originating from the different ontologies, there are some apparent redundancies. Some of those properties always share the same values between them, and some share the same values often.

For now, we prefer to leave all these redundancies in the model, because it is not yet clear which properties are always to get the same value. In time, we will choose whether to remove completely redundant properties in favour of one. For partly redundant properties we could implement a mechanism that ensures redundant properties are synchronized in case one of them is left empty, maintaining the possibility to set those properties to different values where necessary. Here follows an overview of properties that are partly or completely redundant:

Ontology	Property	Value type	Description
Schema.Thing	<i>sameAs</i>	URL	Equal to DC. <i>source</i>
Schema.Thing	<i>url</i>	URL	<p>URL of the webresource about the entity.</p> <p>In case of a Person, this is equal to Schema.Thing.<i>sameAs</i> and DC.<i>source</i>.</p> <p>In case of a content file, this is equal to Schema.CreativeWork.<i>contentUrl</i></p>
Schema.CreativeWork	<i>contentUrl</i>	URL	URL of the image, audio or video file
DC	<i>source</i>	URL	Equal to Schema.Thing. <i>sameAs</i>
Schema.Thing	<i>name</i>	String	Equal to DC. <i>title</i>
DC	<i>title</i>	String	Equal to Schema.Thing. <i>name</i>
DC	<i>creator</i>	String/URL	<p>For types where this makes sense (<b>CreativeWork, Action, Event</b>) the value is the (String) name of the Person/Organization, corresponding to Schema.<i>author/agent/orga nizer</i> related entity.</p> <p>For other types, it is the name/base-URL of the person/organization that</p>

			created the web resource about this entity. In this case, it can equal publisher, or contributor, or both.
DC	<i>publisher</i>	URL	<p>For <b>CreativeWork</b>, this corresponds to the base-URL of the <i>schema_publisher</i>, which is the <b>Organization</b> that published the book, article, score etc.</p> <p>For <b>Products</b>, like vinyl, cd this is the base-URL of the <i>manufacturer</i>.</p> <p>If the thing cannot be published, <i>publisher</i> value is the base-URL of the webresource about this thing. In this case, it can equal creator, or contributor, or both.</p>
DC	<i>contributor</i>	URL	<p>For all types, this value is either the base-URL of the web-resource about the thing, or the full name of the TROMPA <b>User</b> that created this thing.</p> <p>In all cases, it can equal publisher, or contributor, or both.</p>
Schema. <b>MusicComposition</b>	<i>composer</i>	Person	<p>For <b>MusicComposition</b> this value is the same as <i>author</i>.</p> <p>This value can be equal to <i>accountablePerson</i> and/or <i>wasAttributedTo</i>.</p>
Schema. <b>CreativeWork</b>	<i>author</i>	Person	<p>For <b>MusicComposition</b> this value is the same as <i>composer</i>.</p> <p>This value can be equal to <i>accountablePerson</i> and/or</p>

			<i>wasAttributedTo.</i>
Schema. <b>CreativeWork</b>	<i>accountablePerson</i>	Person	To be set with the Person legally accountable for the CreativeWork. Could be left empty if unknown, or could be equal to <i>author</i> and/or <i>composer</i>
PROV-O	<i>wasAttributedTo</i>	URL	<p>For <b>CreativeWork</b> this value is the same as <i>author</i>.</p> <p>For <b>MusicComposition</b> this value is the same as <i>composer</i> and <i>author</i>.</p> <p>For <b>Event</b> this value should be same as <i>organizer</i>.</p> <p>For <b>Action</b> this value should be same as <i>agent</i>.</p> <p>For <b>Product</b> this value should be same as <i>manufacturer</i>.</p> <p>For <b>Organization</b> this value should be same as <i>founder</i>.</p> <p>For other types this property should be left empty, unless the entity is clearly attributable to a <b>Person</b> or <b>Organization</b>.</p>
Schema. <b>Person</b>	<i>birthDate</i>	Date	Is equal to DC. <i>date</i>
Schema. <b>Event</b>	<i>startDate</i>	DateTime	Is equal to DC. <i>date</i>
DC	<i>date</i>	Date	<p>The value represents the date for the thing coming into existence. E.g. the publication date (<b>Article</b>), first performance date (<b>MusicComposition</b>).</p> <p>In case of a <b>Person</b>, it is equal to the <i>birthdate</i>.</p>

			In case of an <b>Event</b> , it is equal to <i>startDate</i>
Schema. <b>Thing</b>	<i>inLanguage</i>	String	<p>On <b>CreativeWork</b>, <b>Event</b> and <b>Action</b> types, this is/are the languages in which the work, event or action is expressed. Multiple languages can be set, like "en,fr,de" in order of priority.</p> <p>This can be the same, or can contain the same value as DC.<i>language</i></p> <p>This value should be left empty if <b>CreativeWork</b> and <b>Event</b> cannot be determined.</p>
DC	<i>language</i>	String	For all types, this language is the language in which the DC properties are written.

**Table 3.3.** Property redundancies guidelines

### 3.4 RDF compatibility guidelines

Currently, all nodes and relations used in the CE internal data model can be traced back to a well-known ontology and have a RDF URI. The aim is to maintain this RDF compatibility throughout the TROMPA project. Optional json-ld output on CE API output is released as part of the CE-API v0.4.0.

By default, all nodes and relations will be automatically labelled with one or more RDF URI's of the ontologies on which the internal data model is based<sup>[not yet implemented]</sup>. These will be available through the type property that is available on all internal model types.

When importing data in the CE, it is possible to add additional RDF URI's to nodes. For this purpose, add one or more comma-separated URI's in the additionalType property available for each internal model type.

```
type: "https://schema.org/VideoObject"
additionalType: "http://purl.org/ontology/mo/MusicalManifestation"
```

This works the same for *additionalProperty*. The mechanism to add custom edges and nodes is RDF compatible.

## 3.5 Ontological guidelines

In order to ensure consistency in the data added to the CE by each partner, we provide concrete guidelines about what fields to set on the object types which we are currently using.

### Person

Use the **CreatePerson** mutation, with the following fields:

- ❖ **contributor** (required): The URL of the institution (e.g. <https://www.upf.edu>), or the URL of a TROMPA contributor (once the CE supports users)
- ❖ **creator** (required): the base url of the website where the information was obtained from (e.g. imslp.org, musicbrainz.org)
- ❖ **description**: A biography of the person if the source contains one, otherwise an empty string.
- ❖ **format** (required): Format of the source where the data for this person came from (this is a mimetype, so for a website this would be something like "text/html")
- ❖ **language** (required): Language of the source of the data, one of en,es,ca,nl,de,fr
- ❖ **publisher**: Set to the same value as 'creator'
- ❖ **source** (required): the URL of the page about this person
- ❖ **subject** (required): Currently set to the dummy string "Composer"
- ❖ **name** (required): The name of the person
- ❖ **title** (required): The name of the person

### MusicComposition

Use the **CreateMusicComposition** mutation, with the following fields

- ❖ **title** (required): Title of the composition
- ❖ **name** (required): Title of the composition
- ❖ **creator** (required): the base url of the website where the information was obtained from (e.g. imslp.org, musicbrainz.org)
- ❖ **creator** (required): Name of the composer (string)
- ❖ **description** (required): currently set to "Composition [name] by [composer]"
- ❖ **source** (required): the URL of the page about this composition
- ❖ **subject** (required): currently set to "[language] Choir piece"
- ❖ **format** (required): Format of the source where the data for this person came from (this is a mimetype, so for a website this would be something like "text/html")
- ❖ **language** (required): Language of the source of the data, one of en,es,ca,nl,de,fr

Add a person as the composer of a composition using the `AddCreativeWorkInterfaceLegalPerson` mutation<sup>1</sup>.

---

<sup>1</sup> There is currently no way to set the 'Composer' of a work, so we set the Author field instead (<https://github.com/trompamusic/ce-api/issues/18>)

```
AddCreativeWorkInterfaceLegalPerson (
  from: {identifier: "composition_id" type: MusicComposition}
  to: {identifier: "composer_id" type: Person}
  field: author
) { }
```

## DigitalDocument

A DigitalDocument represents a data file. It can be linked to another object to say that it is a file of that object (e.g. the score of a composition)

Use the CreateDigitalDocument mutation with the following fields:

- ❖ **contributor**: same as the description for the CreativeWork that this document links to
- ❖ **creator**: same as the description for the CreativeWork that this document links to
- ❖ **description**: same as the description for the CreativeWork that this document links to
- ❖ **format**: the mimetype of this document
- ❖ **Language**: the language that the document is in (if applicable)
- ❖ **source**: the url of the page where this document can be downloaded from
- ❖ **subject**: the same as the subject for the CreativeWork that this document links to
- ❖ **title**: the name of the thing that this document refers to
- ❖ **name**: the name of the thing that this document refers to
- ❖ **relation**: the url of the document on the internet
- ❖ **license**: the license that the document is made available under

## Joining a Document and a Composition

To join a DigitalDocument and a MusicComposition, say that the document is an exampleOfWork of the composition with the following mutation:

```
AddThingInterfaceCreativeWorkInterface(
  from: {identifier: "{document_id}" type:DigitalDocument}
  to: {identifier: "{composition_id}" type:MusicComposition}
  field: exampleOfWork
) { }
```

## Joining Many Documents Together

If there are many documents that represent the same work, they can be joined together using the skos:BroadMatch relation between every pair of documents. Use the following mutation:

```
AddDigitalDocumentBroadMatch(
  from: {identifier: "anid" }
  to: {identifier: "anotherid" }
) { }
```

## Joining the same entity from different sources

If there is metadata from different sources that refer to the same item (person, composition, etc) then use the Add[Object]ExactMatch mutation between all pairs of objects which refer to the same thing.

### 3.6 Private data guidelines [not yet implemented]

Data stored within the CE's graph database is assumed to be public in nature, with the creation and interlinking of open data forming a core focus of the TROMPA project. Nevertheless, certain user data pertaining to TROMPA will need to remain private, or accessible to only particular specified users. Examples include private rehearsal recordings that instrumental players or choir singers may wish to listen to in order to support rehearsal practice, while not necessarily wishing to share them with the rest of the world; rehearsal notes intended for private discussion between a teacher and a student; or, working drafts of scholarly annotations that may need to be iteratively improved and finalised before open publication.

Such requirements will be supported by mandating storage of non-public data in web-accessible locations outside of the CE, tied into the CE only by reference (URI). Fine-grained user-based access authorization can then be implemented at the external stores. The granularity of this integration by reference is likely to be use-case dependent, and remains to be worked out during the course of further development work within the TROMPA project; for instance, externally stored (non-public) annotations may be referenced individually from within the CE, or an externally stored annotation container ("annotations by user X") might be referenced instead.

If a private data item (e.g. a working draft of an annotation) becomes public (is published) during a particular workflow, this can be handled by modifying authorisation at the external storage location accordingly, where integration by reference from the CE is sufficient; or, if the newly-published item is to be discoverable via the `trompa` API, it can be incorporated by value (copied in) to the CE, with the externally hosted working draft referencing the new CE-internal location by URI via the CE's REST-wrapper (section 5).

By use of a shared identity provider between the external stores and the CE, their separation can remain transparent to non-specialist users who simply experience logging into the TROMPA application(s) supporting their use case(s).

Concrete implementations of this authorisation concept remain to be developed at the current stage of the project. Candidate technologies for implementation of the authorised external storage include Solid PODS ("personal online data stores"), and S3 buckets implemented on the Amazon AWS Cognito platform.

Solid<sup>2</sup> is a Web decentralisation project building on a W3C standards-based Linked Data technology stack which aims to enable rich online interactions between users that retain data ownership with each individual user. From a TROMPA perspective, it allows each user to retain fine-grained access control over their personal data, supporting sharing of data with specified users, and simple integration by reference with the CE. Solid PODS are not tied to a monolithic corporate entity, but may rather be obtained from a growing number of public providers, or even be self-hosted by technically savvy users. Further, a TROMPA-hosted provider has been set up for development and testing purposes at MDW and could be opened up to use by a TROMPA audience. This emphasis on user choice and control of web-hosted data provide a pleasing fit to TROMPA's emphasis on FAIR and open data principles.

---

<sup>2</sup> <http://solidproject.org>

Amazon AWS Cognito makes it possible to add user sign-up, sign-in, and access control to web and mobile apps quickly and easily. Amazon Cognito scales to millions of users and supports sign-in with social identity providers, such as Facebook, Google, and Amazon, enterprise identity providers via SAML 2.0 and OpenID Connect. It is integrated with the AWS Identity and Access Management (IAM) service, which makes it possible to manage access to AWS services and resources securely. Using IAM, one can create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources, like for instance S3 buckets and specific directories and objects within them. In the context of the CE it makes it possible to grant access to certain use-case resources to particular users or groups of users, guaranteeing that some user data may be added, modified or deleted only by the user who owns the data, and allowing for sharing data between particular users of the system. A potential benefit of AWS Cognito is their Hosted UI, which eliminates the hosting and secure login security responsibility for the Trompa partners.

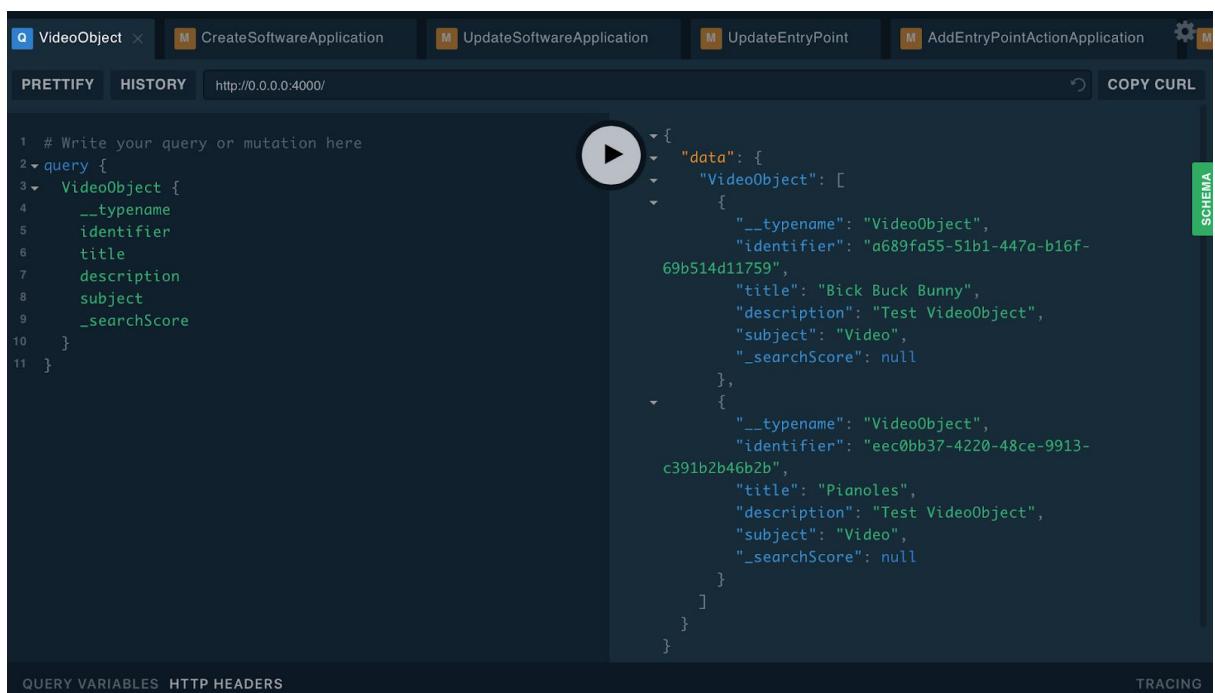
## 4. GraphQL interface for managing data

[GraphQL](#) 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 manual](#) can provide detailed background information for the following sections.

GraphQL supports three types of functionalities that are accessible through the GraphQL API interface:

- ❖ Queries
- ❖ Mutations
- ❖ Subscriptions

For the CE API, a [graphic interface](#) is available that provides a human-friendly way to interact with the GraphQL API interface and supports rich introspection of the schema.



The screenshot shows a GraphQL interface with the following components:

- Top Bar:** Buttons for VideoObject, CreateSoftwareApplication, UpdateSoftwareApplication, UpdateEntryPoint, AddEntryPointActionApplication, and a gear icon.
- Header:** PRETTIFY, HISTORY, http://0.0.0.0:4000/, COPY CURL.
- Left Panel (Query Editor):** A code editor containing a GraphQL query:

```
1 # Write your query or mutation here
2 query {
3   VideoObject {
4     __typename
5     identifier
6     title
7     description
8     subject
9     _searchScore
10  }
11 }
```
- Middle Panel (Response Preview):** A large text area showing the JSON response after clicking the play button. The response is:

```
{
  "data": {
    "VideoObject": [
      {
        "__typename": "VideoObject",
        "identifier": "a689fa55-51b1-447a-b16f-69b514d11759",
        "title": "Bick Buck Bunny",
        "description": "Test VideoObject",
        "subject": "Video",
        "_searchScore": null
      },
      {
        "__typename": "VideoObject",
        "identifier": "eec0bb37-4220-48ce-9913-c391b2b46b2b",
        "title": "Pianoles",
        "description": "Test VideoObject",
        "subject": "Video",
        "_searchScore": null
      }
    ]
  }
}
```
- Right Panel (Schema):** A green bar labeled "SCHEMA".
- Bottom Navigation:** QUERY VARIABLES, HTTP HEADERS, TRACING.

Figure 4.1

The top part of the interface is for query management. The left part is used to help composing a request and the right part will show the response after clicking the > button.

## 4.1 Schema introspection

On the far right of the interface there is a green ‘SCHEMA’ tab that offers a complete overview of the type and property schema underlying the CE database. It also offers an overview for all the ‘custom’ functionalities available, like adding, mutating or deleting specific node relations.

The screenshot shows the Neo4j Cypher browser interface. On the left, there is a code editor window with a query:1 # Write your query or mutation here
2
3 query {
4 VideoObject {
5 \_\_typename
6 identifier
7 title
8 description
9 subject
10 \_searchScore
11 }
12 }Below the code editor are two buttons: 'QUERY VARIABLES' and 'HTTP HEADERS'. On the right side of the interface, there is a sidebar titled 'SCHEMA' which contains a search bar and a list of schema elements. One element, 'VideoObject', is selected. To the right of the sidebar, there is a detailed view of the 'VideoObject' schema. This view includes the URL 'http://dublincore.org/2012/06/14/dcelements/o/https://schema.org/CreativeWork,https://', a 'TYPE DETAILS' section listing various interfaces implemented by 'VideoObject' (e.g., 'type VideoObject', 'implements MetadataInterface', etc.), and a list of properties with their types (e.g., 'contributor: String!', 'coverage: String', 'creator: String!', 'date: \_Neo4jDate').

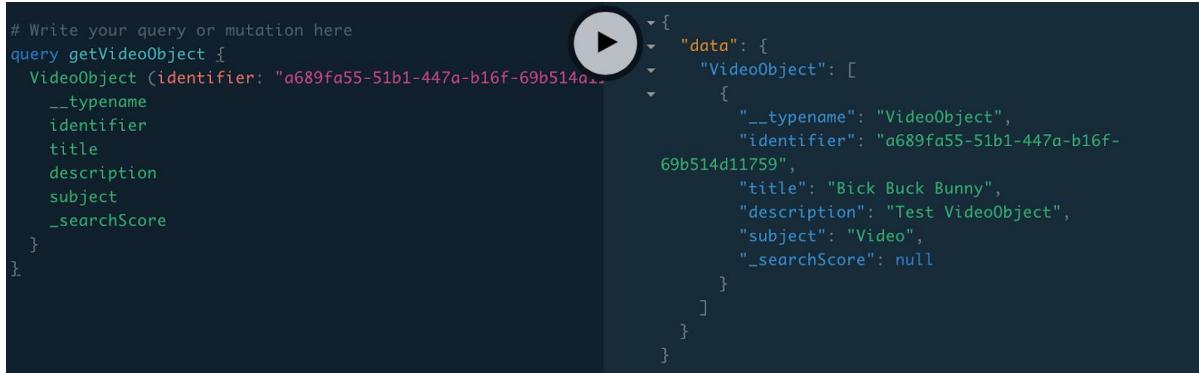
Figure 4.2

This introspection tool shows the actual schema and its possibilities, and is the best way to explore the schema and how to access the CE api functionalities.

## 4.2. Queries

Queries are requests for existing data from the database.

### 4.2.1 Simple query for one node



The screenshot shows a GraphQL playground interface. On the left, there is a code editor containing a GraphQL query:# Write your query or mutation here
query getVideoObject {
 VideoObject (identifier: "a689fa55-51b1-447a-b16f-69b514d11759") {
 \_\_typename
 identifier
 title
 description
 subject
 \_searchScore
 }
}A play button icon is positioned above the code editor. To the right, the results of the query are displayed as a JSON object:{
 "data": [
 {
 "VideoObject": [
 {
 "\_\_typename": "VideoObject",
 "identifier": "a689fa55-51b1-447a-b16f-69b514d11759",
 "title": "Bick Buck Bunny",
 "description": "Test VideoObject",
 "subject": "Video",
 "\_searchScore": null
 }
 ]
 }
 ]
}

Figure 4.3

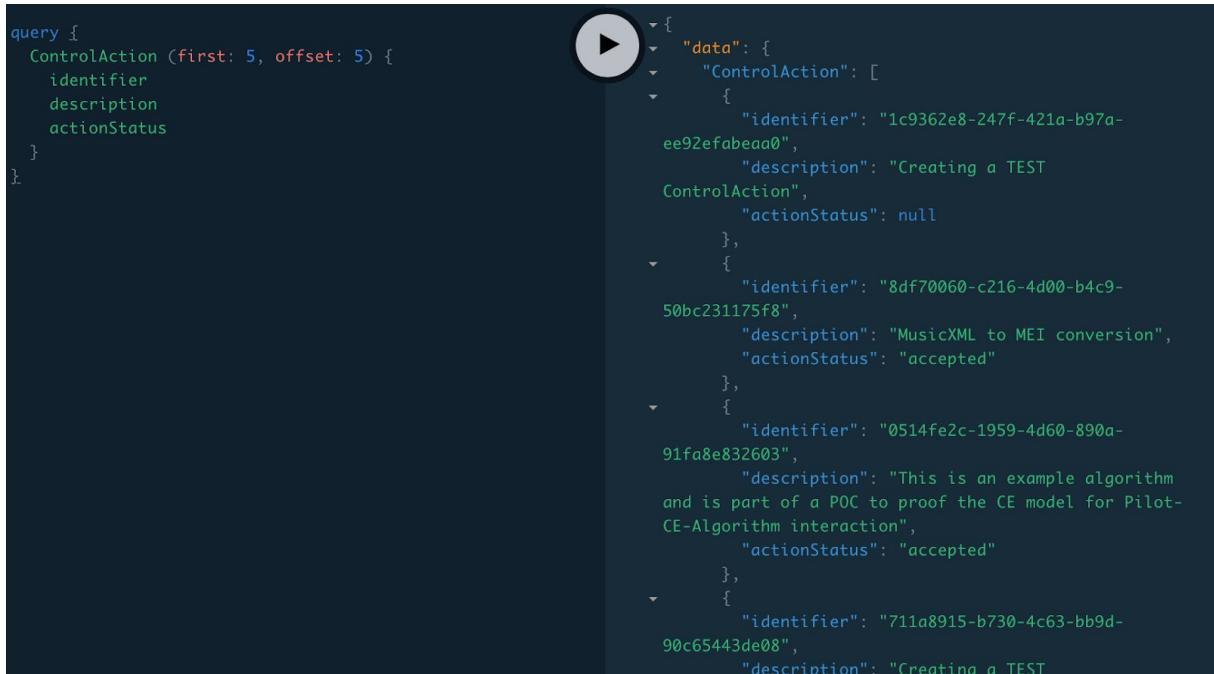
A query starts with the phrase ‘query’ and typically consists of:

- ❖ 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

#### 4.2.2 Simple query for multiple nodes



The screenshot shows a GraphQL playground interface. On the left, there is a code editor containing the following GraphQL query:

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

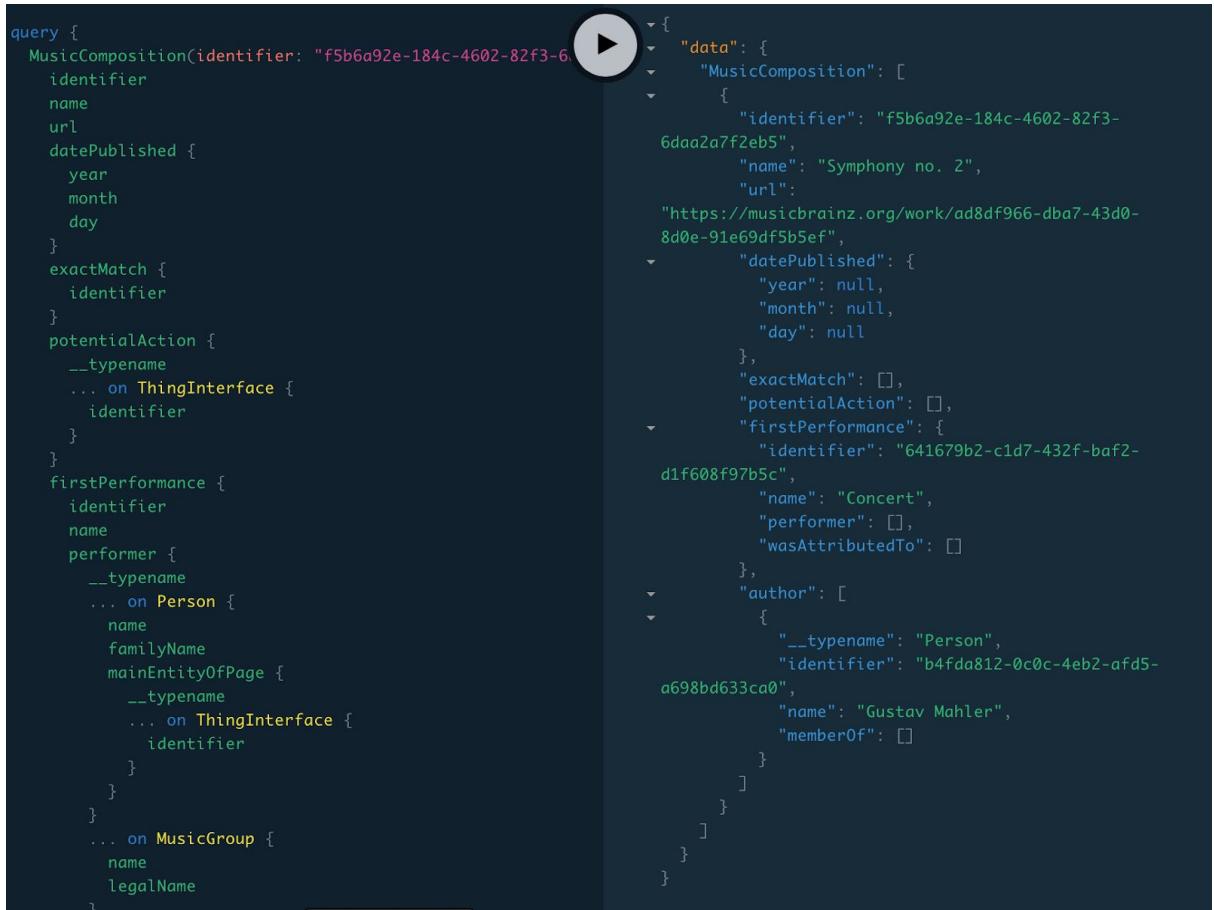
On the right, the results are displayed in a tree-view format. A play button icon is visible at the top left of the results area.

```
{
  "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"
      }
    ]
  }
}
```

Figure 4.4

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

### 4.2.3 Complex query



The screenshot shows a GraphQL playground interface. On the left, there is a code editor containing a complex GraphQL query. On the right, there is a results panel displaying the JSON response generated by the query. The query itself is quite detailed, involving multiple types like `MusicComposition`, `ThingInterface`, `Person`, and `MusicGroup`. The results panel shows a hierarchical structure of data, starting with a top-level object that includes properties for `data` and `MusicComposition`. The `MusicComposition` array contains one item, which is a nested object with properties like `identifier`, `name`, `url`, and `datePublished`. The `datePublished` field is expanded to show null values for `year`, `month`, and `day`. Other fields like `exactMatch` and `potentialAction` are also present. The `firstPerformance` field is expanded to show a nested object with properties for `identifier`, `name`, and `performer`. The `performer` field is expanded to show a nested object with properties for `__typename`, `name`, `familyName`, and `mainEntityOfPage`. The `mainEntityOfPage` field is expanded to show a nested object with properties for `__typename`, `identifier`, and `name`. The `name` field is expanded to show a nested object with properties for `__typename`, `identifier`, and `name`. The `name` field is further expanded to show a nested object with properties for `__typename`, `identifier`, and `name`.

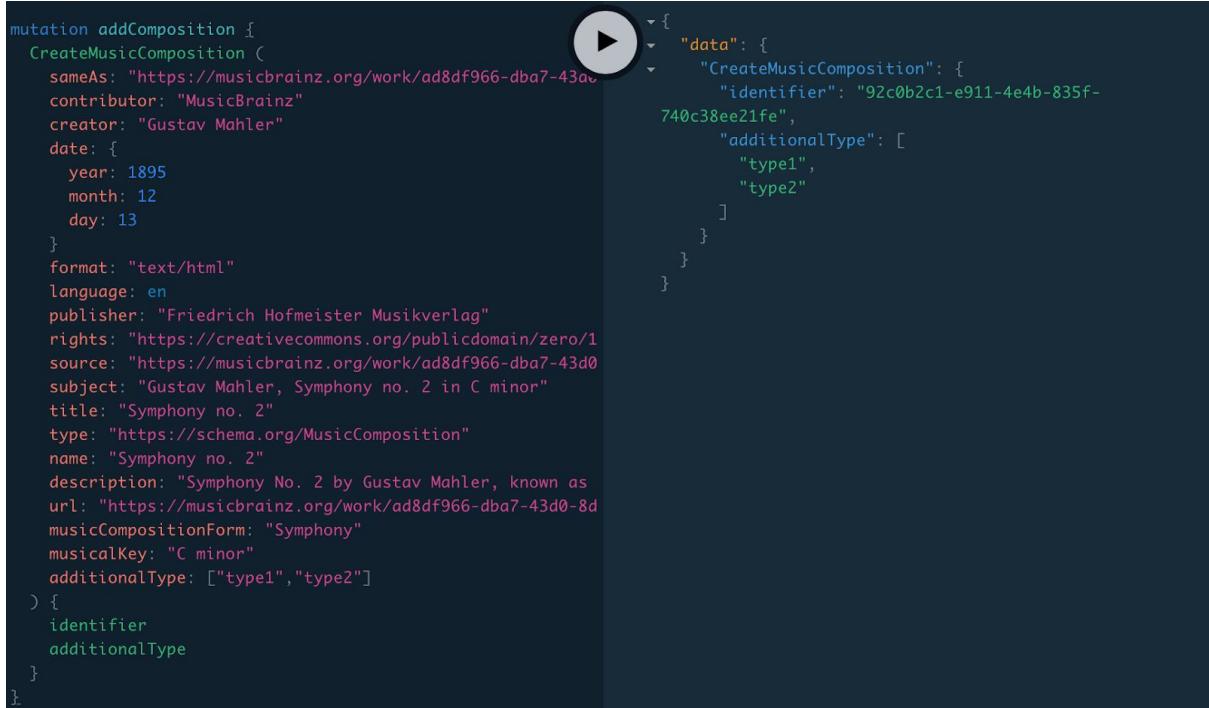
Figure 4.5

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.

## 4.3. Mutations

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

### 4.3.1 Creating a node



The screenshot shows a GraphQL mutation for creating a music composition. The mutation is named 'addComposition' and takes a single argument 'CreateMusicComposition'. This argument contains various properties of the composition, such as 'sameAs', 'contributor', 'creator', 'date' (with year, month, and day), 'format', 'language', 'publisher', 'rights', 'source', 'subject', 'title', 'type', 'name', 'description', 'url', 'musicCompositionForm', 'musicalKey', and 'additionalType'. The 'additionalType' field is shown as an array containing 'type1' and 'type2'. After the argument, there is a block brace {}, followed by a colon, then 'identifier' and 'additionalType' listed. The code is syntax-highlighted with colors for different types of tokens.

```
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
  }
}
```

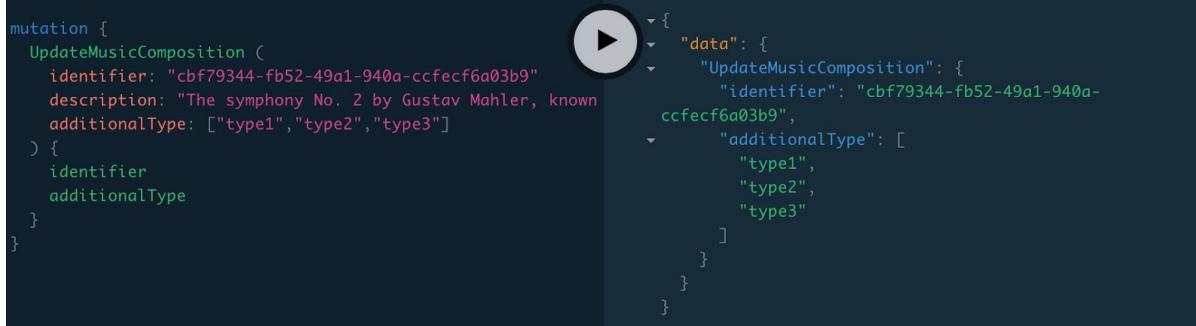
Figure 4.6

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.

#### 4.3.2 Updating a node



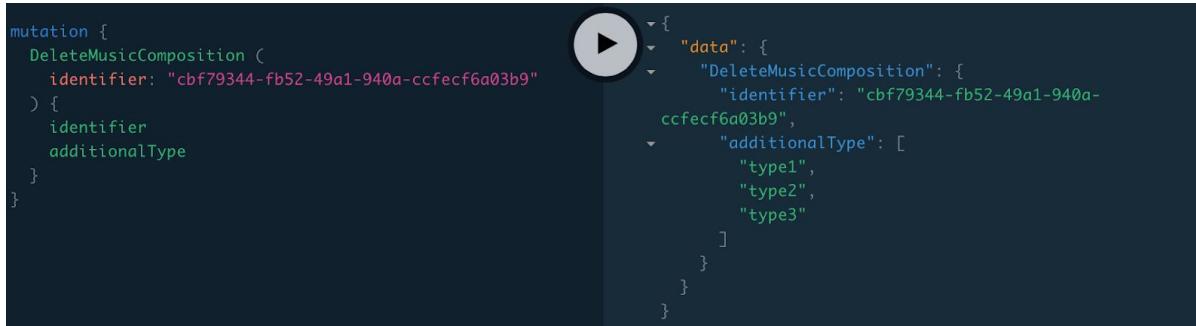
```
mutation {
  UpdateMusicComposition (
    identifier: "cbf79344-fb52-49a1-940a-ccfecf6a03b9"
    description: "The symphony No. 2 by Gustav Mahler, known"
    additionalType: ["type1", "type2", "type3"]
  ) {
    identifier
    additionalType
  }
}
```

The screenshot shows a GraphQL mutation named 'UpdateMusicComposition'. It takes three arguments: 'identifier' (set to 'cbf79344-fb52-49a1-940a-ccfecf6a03b9'), 'description' (set to 'The symphony No. 2 by Gustav Mahler, known'), and 'additionalType' (set to an array containing 'type1', 'type2', and 'type3'). The response on the right side of the playground shows the updated node with its identifier, description, and the full array of 'additionalType'.

Figure 4.7

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.

#### 4.3.4 Deleting a node



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

The screenshot shows a GraphQL mutation named 'DeleteMusicComposition'. It takes one argument: 'identifier' (set to 'cbf79344-fb52-49a1-940a-ccfecf6a03b9'). The response on the right side of the playground shows the deleted node with its identifier and the full array of 'additionalType'.

Figure 4.8

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:

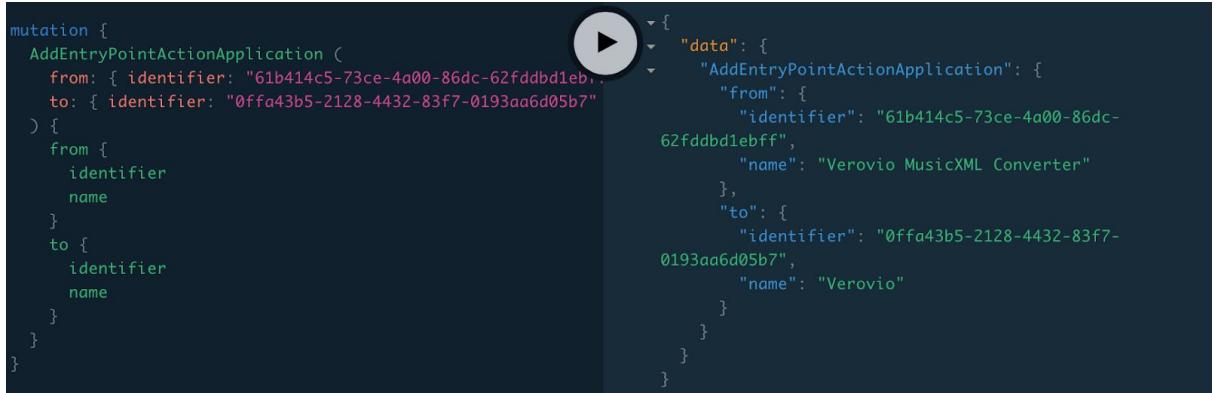


```
mutation {
  DeleteMusicComposition (
    identifier: "cbf79344-fb52-49a1-940a-ccfecf6a03b9"
  ) {
    identifier
    additionalType
  }
}
```

The screenshot shows a second GraphQL mutation named 'DeleteMusicComposition' with the same identifier as the previous one. The response on the right side of the playground shows the deleted node with its identifier and the full array of 'additionalType'.

Figure 4.9

#### 4.3.5 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
    }
  }
}
```

The screenshot shows a GraphQL playground interface. On the left, there is a code editor containing the provided GraphQL mutation. On the right, there is a results panel displaying the JSON response to the mutation. The response includes a play button icon, a "data" field, and an "AddEntryPointActionApplication" field which contains "from" and "to" objects with their respective identifiers and names.

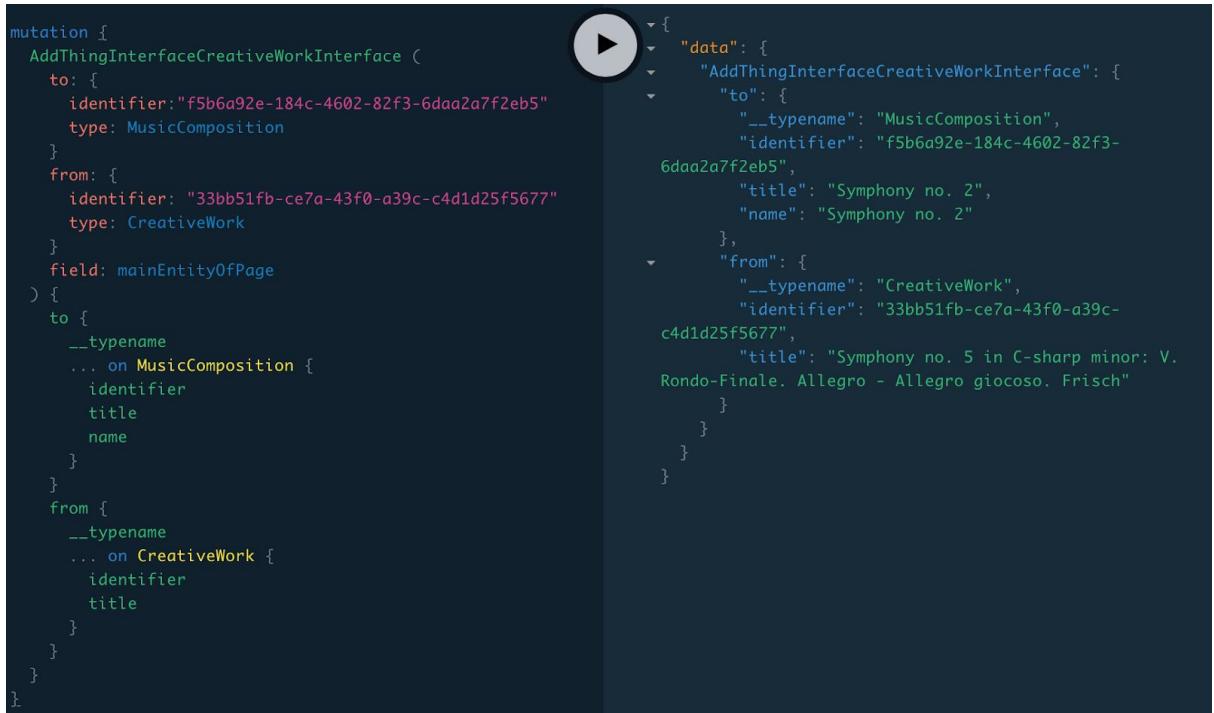
```
{
  "data": {
    "AddEntryPointActionApplication": {
      "from": {
        "identifier": "61b414c5-73ce-4a00-86dc-62fddbd1ebff",
        "name": "Verovio MusicXML Converter"
      },
      "to": {
        "identifier": "0ffa43b5-2128-4432-83f7-0193aa6d05b7",
        "name": "Verovio"
      }
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

Figure 4.10

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.

#### 4.3.6 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
      }
    }
  }
}
```

The screenshot shows a GraphQL playground interface. On the left, there is a code editor containing the provided GraphQL mutation. On the right, there is a results panel displaying the JSON response to the mutation. The response includes a play button icon, a "data" field, and an "AddThingInterfaceCreativeWorkInterface" field which contains "to" and "from" objects. The "to" object is a "MusicComposition" with identifier "f5b6a92e-184c-4602-82f3-6daa2a7f2eb5" and title "Symphony no. 2". The "from" object is a "CreativeWork" with identifier "33bb51fb-ce7a-43f0-a39c-c4d1d25f5677" and title "Symphony no. 5 in C-sharp minor: V. Rondo-Finale. Allegro - Allegro giocoso. Frisch".

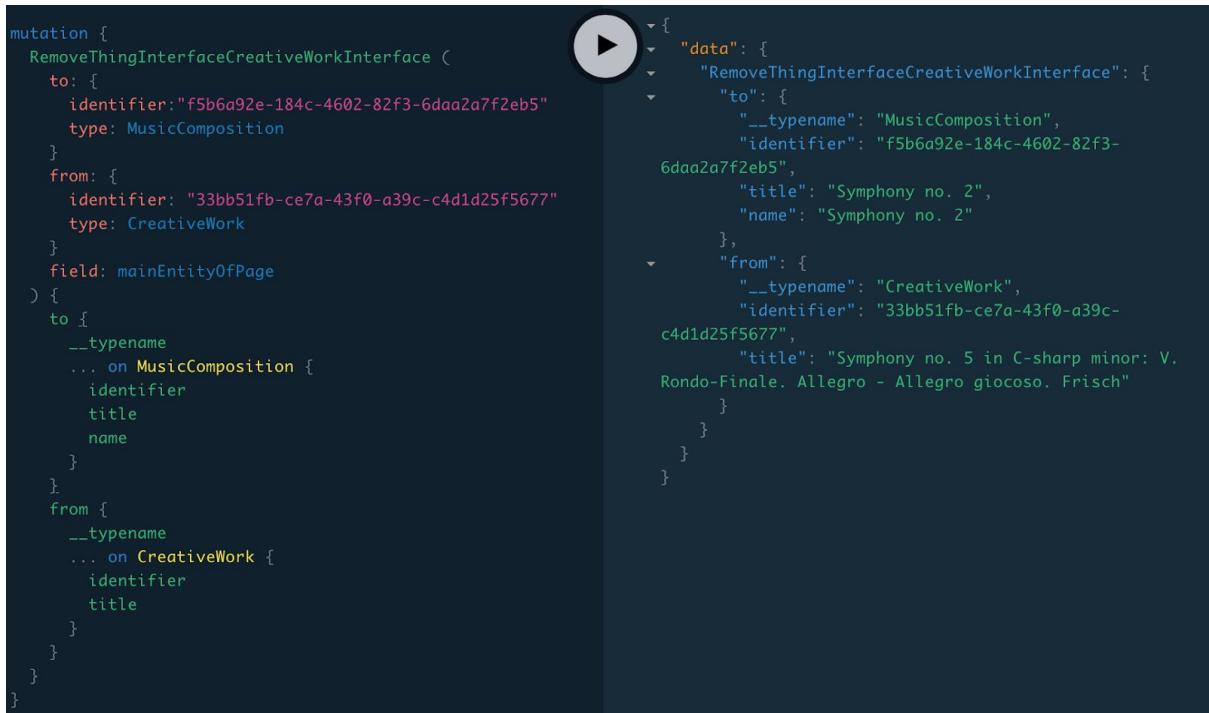
```
{
  "data": {
    "AddThingInterfaceCreativeWorkInterface": {
      "to": {
        "__typename": "MusicComposition",
        "identifier": "f5b6a92e-184c-4602-82f3-6daa2a7f2eb5",
        "title": "Symphony no. 2",
        "name": "Symphony no. 2"
      },
      "from": {
        "__typename": "CreativeWork",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

Figure 4.11

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.

#### 4.3.7 Remove a relation between nodes



The screenshot shows a GraphQL playground interface. On the left, there is a code editor containing a mutation query. On the right, there is a results panel showing the response from the GraphQL server.

**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"**

**}**

**}**

**}**

Figure 4.12

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 .

## 4.4. Subscriptions [not yet implemented]

Some algorithms that we expect to run within the CE will be run by a partner on all documents that exist in the CE. As an example, MTG might want to run MFCC calculations on all audio files added to the CE, regardless of who adds these documents. To facilitate this, specific Subscriptions will be added. The subscriptions will not trigger the ControlActions itself, but it will allow partners to trigger ControlActions on certain actions. For example, the Subscription will allow applications to listen when a DigitalDocument gets added. A separate process, which is subscribed to this topic, gets the notification and will trigger a ControlAction so again a separate process/algorithm can run its task(s).

## 4.5 Authentication [not yet implemented]

The current version (0.4) of the CE-API GraphQL interface does not yet support authentication. This means that all data can be accessed, changed and added by anybody. When the (hosted) CE-API is made available to the general public, access control and authentication will be required. At first all partners of the project will be provided with tokens with write access to the CE-API. Each token can be allocated allowed actions e.g. queries, subscriptions and mutations. At a later stage third parties can be provided with tokens with write access to specific actions.

# 5. REST interface

The CE api provides a very basic REST interface. The main purpose of this REST interface is to provide a unique URL for each node in the CE database and to provide JSON-LD output:

[https://\[ce-api-domain\]/\[node UUID\]](https://[ce-api-domain]/[node UUID])

Adjacent nodes in the graph which are related to the one requested will be included in the response only by reference to their respective REST URL. They will not be embedded in the response, i.e. their content will not be returned (until their own REST URL is requested in due course). Queries that require the embedding of related nodes' contents must instead be formulated using the GraphQL interface.

A GET request to this URL will return the node object in json format, including properties that contain a value. Properties consisting of a relation to other nodes do not contain deeper json objects, but will contain the URL (CE API REST interface) or URLs to related node(s)

```
{
  "identifier": "88f82c5f-4a4f-4218-a395-6458443112a4",
  "image": null,
  "nodeValue": "http://api.trompamusic.eu/33bb51fb-ce7a-43f0-a39c-c4d1d25f5677",
  "additionalType": null,
  "valueReference": "CreativeWork",
  "name": "A music recording",
  "description": "This should be an existing TROMPA reference of a VideoObject or AudioObject type",
```

```

"alternateName": null,
"title": "A music recording",
"type": null,
"propertyID": "644b462f-35f3-4ee1-8204-82c98735fbb0",
"value": null
}

```

RESTful POST, PUT, PATCH and DELETE requests will not be available, and will be covered by functionalities provided through the GraphQL interface.

## 6. Integration of jobs and processes

As described in D5.1 Data Infrastructure and D5.3 TROMPA Processing Library, Music Information Retrieval (MIR) technologies as developed in WP3, and Crowd-powered improvements as developed in WP4 are ultimately to be integrated with the CE. The provisioned functionalities of the CE, as described and demonstrated in the previous chapters, allow for this integration; The CE data model in combination with the CE GraphQL interface enables Component and ultimately (pilot) application developers to create nodes in the CE database that could serve as jobs for WP3 and WP4 technologies to be picked up and processed.

In turn, WP3 and WP4 developers can set up a system to retrieve those jobs, to be executed against data referenced in the CE. After completion of the job, references to the results can be written back as nodes in the CE database. Subsequently, relations can be created between those results and the larger TROMPA data set. Taken together, these functionalities constitute an important part of the integration envisioned for WP5 and create a technical basis for the crowd powered enrichment as envisioned for the TROMPA project.

This approach demands quite some coordination between (pilot) application, Component and WP3/4 developers and a likely outcome is different approaches and practices for each combination of pilot/Component and WP3/4 technology and duplication of efforts. Some developers might be tempted to leave the CE out of the loop and create direct access between Component and WP3/4 technologies, putting aside the chance to enrich the TROMPA dataset with user generated content on the basis of public data contained in the CE.

To mitigate and minimize this risk, and to make it easier for TROMPA participants and 3rd party developers to contribute rich functionalities to the TROMPA project in a scalable way, a generic solution is created for Component-CE-WP3/4 integration. This solution, as presented in this chapter, provides a standardised method for task/job creation and retrieval and allows both Components and WP3/4 systems to handle jobs in real time or in batches in asynchronous fashion. This not only addresses the needs of most Component and WP3/4 developers to integrate their work with the CE, but also ensures that WP3/4 produced data is contributed to the TROMPA dataset.

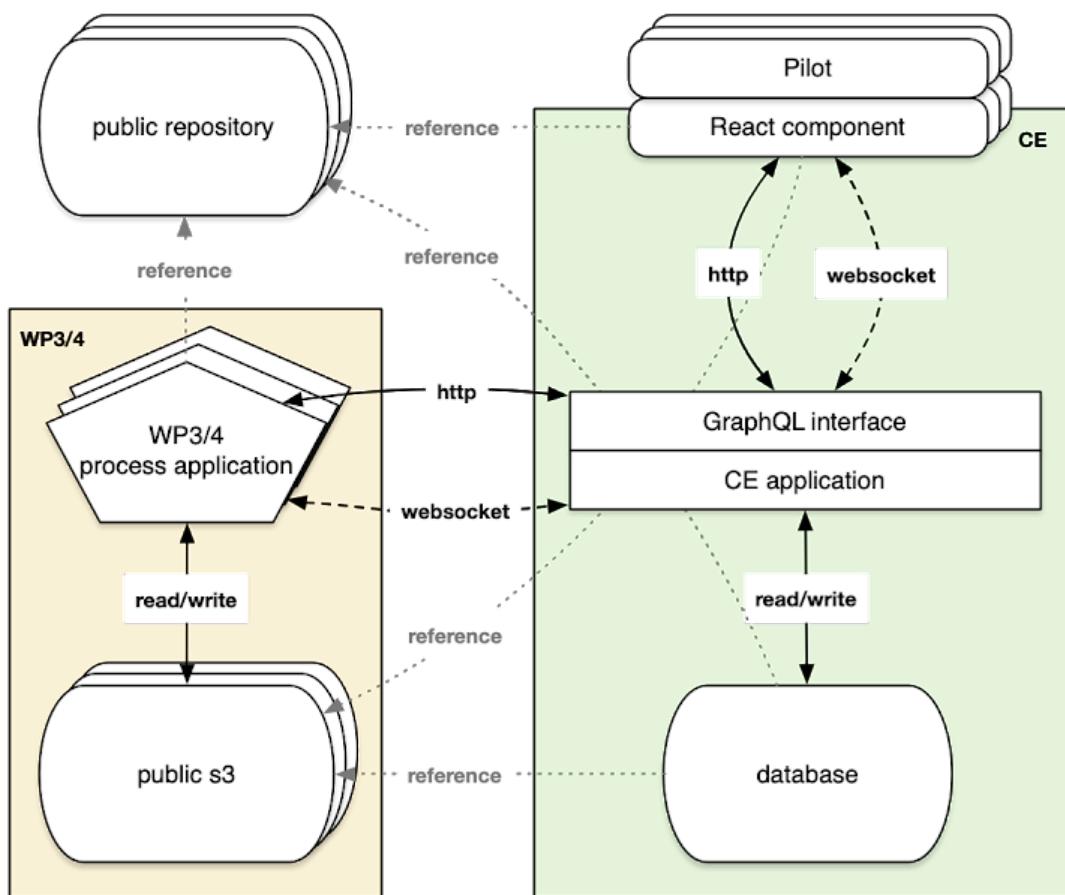
### 6.1 Generic solution overview

At its most basic, the process to be automated is as follows:

- ❖ Component user chooses target content, referenced in CE database

- ❖ Component user creates a job to run a process on this content
- ❖ Process picks up job
- ❖ Process executes job on target content, creating and storing a result
- ❖ Process writes reference to result in CE database
- ❖ Component picks up result
- ❖ Component user consumes result

A [subscription](#) mechanism, as described in section 4.4, can enable both Component and Process system to be actively updated on job creation and status updates in real time (or by http polling). This way, the CE becomes the intermediary of Component-WP3/4 interactions. This offers a standardized solution for integration and ensures WP3/4 produced data is referenced and gets interlinked with the larger TROMPA dataset.



**Figure 6.1**

## 6.2 Data model

The generic Component-CE-WP3/4 interaction solution is based on a [schema.org](#) compatible data model that can be broken down into three parts:

- ❖ Template nodes - maintained by WP3/4 developers - used by Component(s)
- ❖ Instance nodes - created by Component(s), maintained by CE
- ❖ Public nodes - representing the (public) content on which the WP3/4 process is done and the results

### 6.2.1 Template nodes

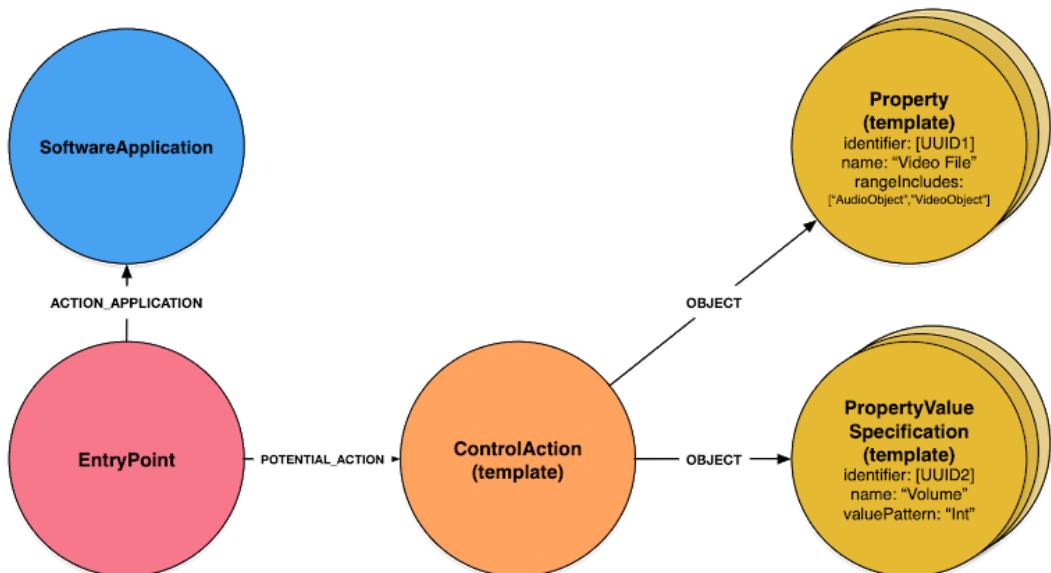


Figure 6.2

Each WP3/4 process application, e.g. an alignment tool, will need a [SoftwareApplication](#) 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 a [EntryPoint](#) node. This entry point corresponds to a user interface that enables a user 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](#) property.

The [ControlAction](#) 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](#) nodes and relating them to the [ControlAction](#) through the *object* property. Required parameters that 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](#) node through the same *object* property.

Together, these **EntryPoint**, **ControlAction**, **PropertyValueSpecification** and **Property** nodes determine what the end user 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.

### 6.2.2 Instance nodes

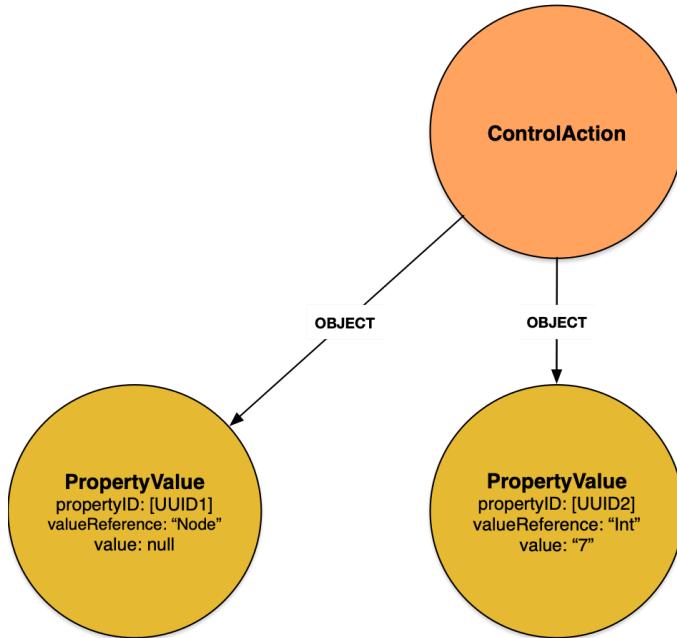


Figure 6.3

The CE GraphQL interface exposes a predefined mutation (**RequestControlAction**) that will create a set of nodes based on the ‘template’ as presented in the previous chapter 3.2.1. In effect, the nodes created by this request instantiate a ‘job’ in the CE database that can now be acted on, followed and updated.

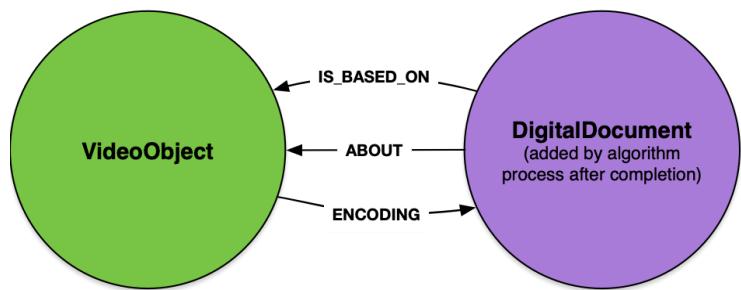
If the **RequestControlAction** request passes validation, it will create a **ControlAction** node that is a copy of the ‘template’, plus one or more **PropertyValue** nodes, derived from the template property nodes, that contain the parameters needed to execute the algorithm process. The thus created **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.

### 6.2.3 Public nodes

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.



**Figure 6.4**

#### 6.2.4 End result

After a successful request-job-result cycle, the final constellation of nodes would look like this:

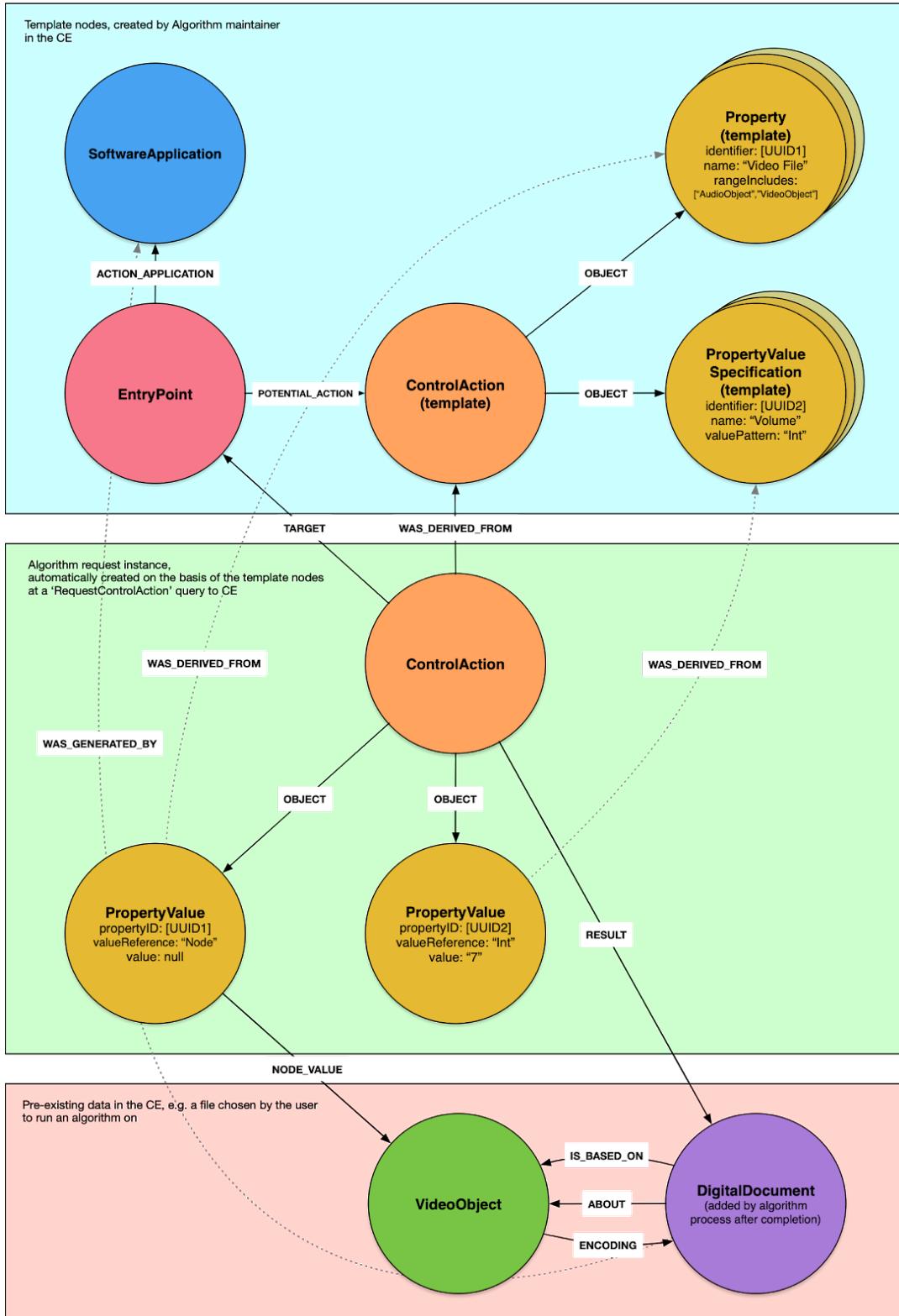


Figure 6.5

## 6.3 Perspective of algorithm process application

For algorithm process maintainers (WP3/4), the first responsibility would be to enter the correct ‘template’ nodes into the CE database. If entered correctly, a Component can query for available algorithm processes with GraphQL queries for **EntryPoints** (potential ‘jobs’). An **EntryPoint** and related **Property** and **PropertyValueSpecification** nodes should contain enough information to dynamically set up a UI for a job request.

With this **EntryPoint** set up, the algorithm process application can now detect whether a job is requested. It can regularly query the CE database for new **ControlActions** derived from the ‘template’ **ControlAction**. Or it could subscribe to the creation of **such ControlActions** with a custom ‘ControlActionRequest’ GraphQL subscription, and be immediately notified of a new request over a websocket connection.

After a new request came in, the algorithm process application can then retrieve the necessary parameters and file(s) to act on and start writing back status or error updates on the **ControlAction** node that represents the job request.

After the completion of the process, the result file(s) can be written to a public repository. The URL to this result file can then be added to the CE database and the status of the **ControlAction** updated to ‘complete’.

The result can now be consumed by the user and can be found through the CE GraphQL interface. The algorithm process application can enrich the TROMPA dataset further by adding additional relations between result file and source file references, as well as to any other relevant nodes in the CE database. Those relations could for example be a *about* or *encoding* relation between result and source, or a *generatedBy* relation to the **SoftwareApplication**. This would greatly improve the chances that subsequent users find and re-use the result file.

### 6.3.1 GraphQL queries

Following are examples of the GraphQL queries relevant for maintainers of algorithm process applications. The examples were made using the default [playground](#) environment, which provides rich features for experimenting with and test GraphQL queries. [Apollo](#) offers a number of software libraries that can help integrate GraphQL in an application.

#### 6.3.1.1 Create and maintain template nodes

This section corresponds to the node data model presented in 6.2.1

### Create SoftwareApplication

```
mutation {
  CreateSoftwareApplication (
    contributor: "https://www.verovio.org"
    title: "Verovio MusicXML Converter"
    name: "Verovio MusicXML Converter"
    creator: "Verovio"
    description: "Verovio supports conversion from MusicXML to"
    source: "https://github.com/rism-ch/verovio"
    subject: "Music notation engraving library for MEI with Mu"
    format: "html"
    language: en
  ) {
    identifier
  }
}
```

Figure 6.6

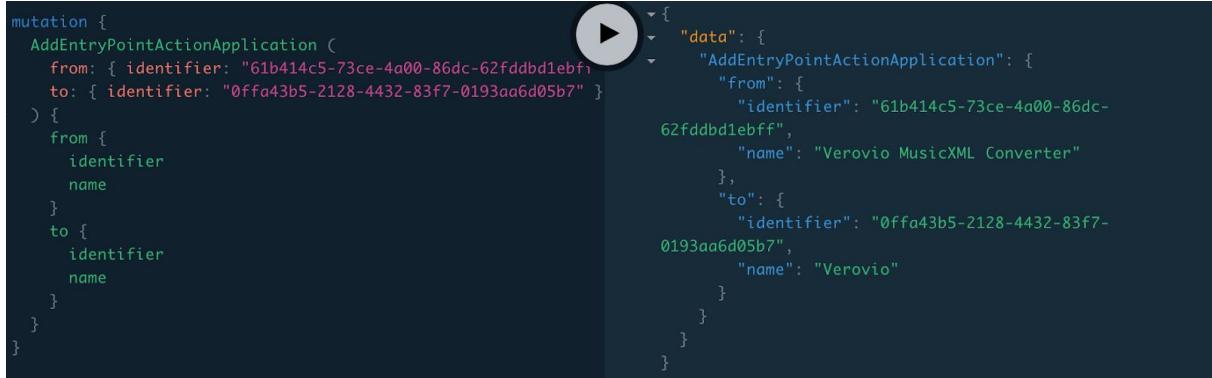
This node will allow to group a number of **EntryPoints** under the same name. For example, the Verovio software bundle offers a number of commands to run. Each Verovio command to be made accessible to Component users should have its own **EntryPoint**, each related to the Verovio **SoftwareApplication** node through the *actionApplication* property. Adding metadata fields will help users searching for available functionalities of a specific software package.

### Create EntryPoint

```
mutation {
  CreateEntryPoint (
    contributor: "https://www.verovio.org"
    title: "Verovio MusicXML Converter"
    name: "Verovio MusicXML Converter"
    creator: "Verovio"
    description: "Verovio supports conversion from MusicXML to"
    source: "https://github.com/rism-ch/verovio"
    subject: "Music notation engraving library for MEI with Mu"
    format: "html"
    language: en
    actionPlatform: "TROMPA Algorithm Proof-Of-Concept"
    contentType: ["json"]
    encodingType: ["text"]
  ) {
    identifier
  }
}
```

Figure 6.7

Create *actionApplication* relation between **SoftwareApplication** and **EntryPoint**



```

mutation {
  AddEntryPointActionApplication (
    from: { identifier: "61b414c5-73ce-4a00-86dc-62fddbd1ebff" }
    to: { identifier: "0ffa43b5-2128-4432-83f7-0193aa6d05b7" }
  ) {
    from {
      identifier
      name
    }
    to {
      identifier
      name
    }
  }
}

```

The screenshot shows a GraphQL mutation named 'AddEntryPointActionApplication'. It takes two arguments: 'from' and 'to'. Both arguments are objects with 'identifier' and 'name' fields. The 'from' object has an identifier of '61b414c5-73ce-4a00-86dc-62fddbd1ebff' and a name of 'Verovio MusicXML Converter'. The 'to' object has an identifier of '0ffa43b5-2128-4432-83f7-0193aa6d05b7' and a name of 'Verovio'.

Figure 6.8

### Create (template) ControlAction



```

mutation {
  CreateControlAction (
    description: "MusicXML to MEI conversion"
    name: "MusicXML to MEI conversion"
    actionStatus: accepted
  ) {
    identifier
    description
    actionStatus
  }
}

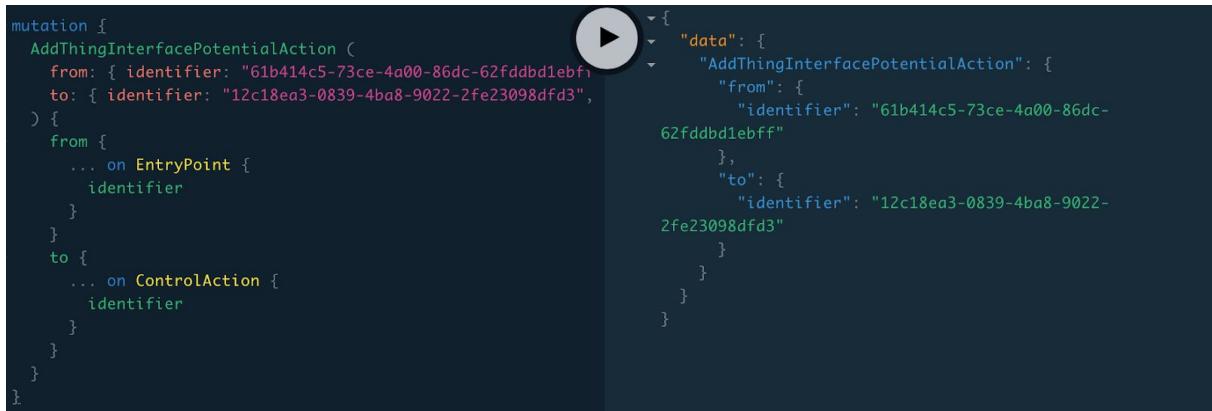
```

The screenshot shows a GraphQL mutation named 'CreateControlAction'. It takes three arguments: 'description' (set to 'MusicXML to MEI conversion'), 'name' (set to 'MusicXML to MEI conversion'), and 'actionStatus' (set to 'accepted'). The returned object has 'identifier', 'description', and 'actionStatus' fields.

Figure 6.9

This **ControlAction** node will be the model for the ‘job’ created when a user does an 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. The default *actionStatus* for a newly instantiated **ControlAction** ‘job’ can be set here.

### Create potentialAction relation between EntryPoint and (template) ControlAction



```

mutation {
  AddThingInterfacePotentialAction (
    from: { identifier: "61b414c5-73ce-4a00-86dc-62fddbd1ebff" }
    to: { identifier: "12c18ea3-0839-4ba8-9022-2fe23098dfd3" }
  ) {
    from {
      ... on EntryPoint {
        identifier
      }
    }
    to {
      ... on ControlAction {
        identifier
      }
    }
  }
}

```

The screenshot shows a GraphQL mutation named 'AddThingInterfacePotentialAction'. It takes two arguments: 'from' and 'to'. Both arguments are objects with 'identifier' fields. The 'from' object has an identifier of '61b414c5-73ce-4a00-86dc-62fddbd1ebff'. The 'to' object has an identifier of '12c18ea3-0839-4ba8-9022-2fe23098dfd3'.

Figure 6.10

#### Create **Property** (template for a relation parameter for a CE reference to a source file)

```
mutation {
  CreateProperty (
    title: "MusicXML File"
    name: "targetFile"
    description: "Select a MusicXML file to be converted to MI"
    rangeIncludes: [DigitalDocument]
  ) {
    identifier
  }
}
```

Figure 6.11

Each template **ControlAction** will probably have at least one parameter that will act as a pointer to an existing node in the CE database, probably referencing a content file at some public repository. The *rangeIncludes* property accepts an array of possible node types for this content reference and can be used by the Component developer to limit the type of nodes (content types) that can be selected.

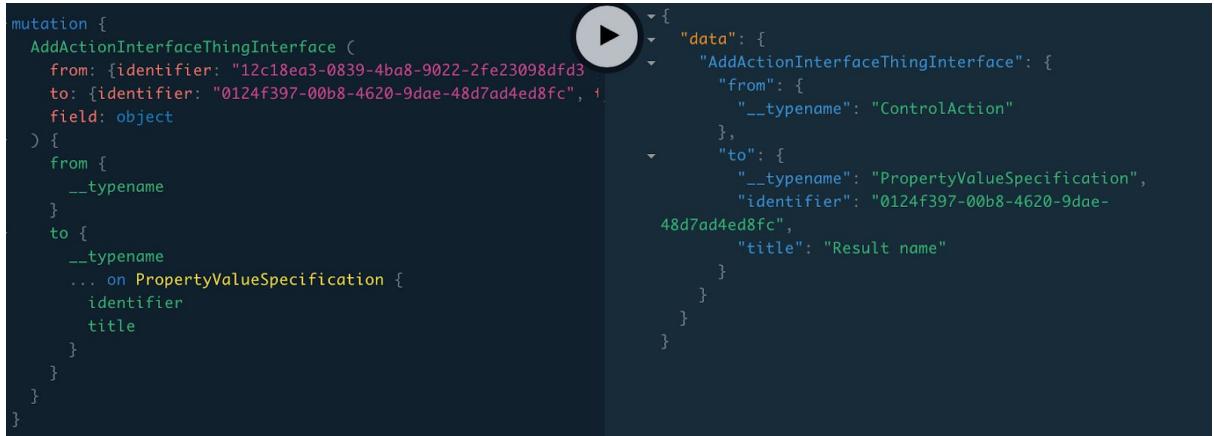
#### Create **PropertyValueSpecification** (template for a scalar parameter)

```
mutation {
  CreatePropertyValueSpecification (
    title: "Result name"
    name: "Result name"
    description: "What name would you like to give the result?"
    defaultValue: ""
    valueMaxLength: 100
    valueMinLength: 4
    multipleValues: false
    valueName: "resultName"
    valuePattern: String
    valueRequired: true
  ) {
    identifier
  }
}
```

Figure 6.12

Each **PropertyValueSpecification** defines a scalar input parameter that the Component user should be prompted with when preparing the request for an algorithm process job. There are numerous properties that can be used to set requirements, type and limits for a scalar parameter. With these properties, a Component developer can set up the input field for this parameter.

Create *object* relation between (template) **ControlAction** and **PropertyValueSpecification** (similar to relation to **Property**)



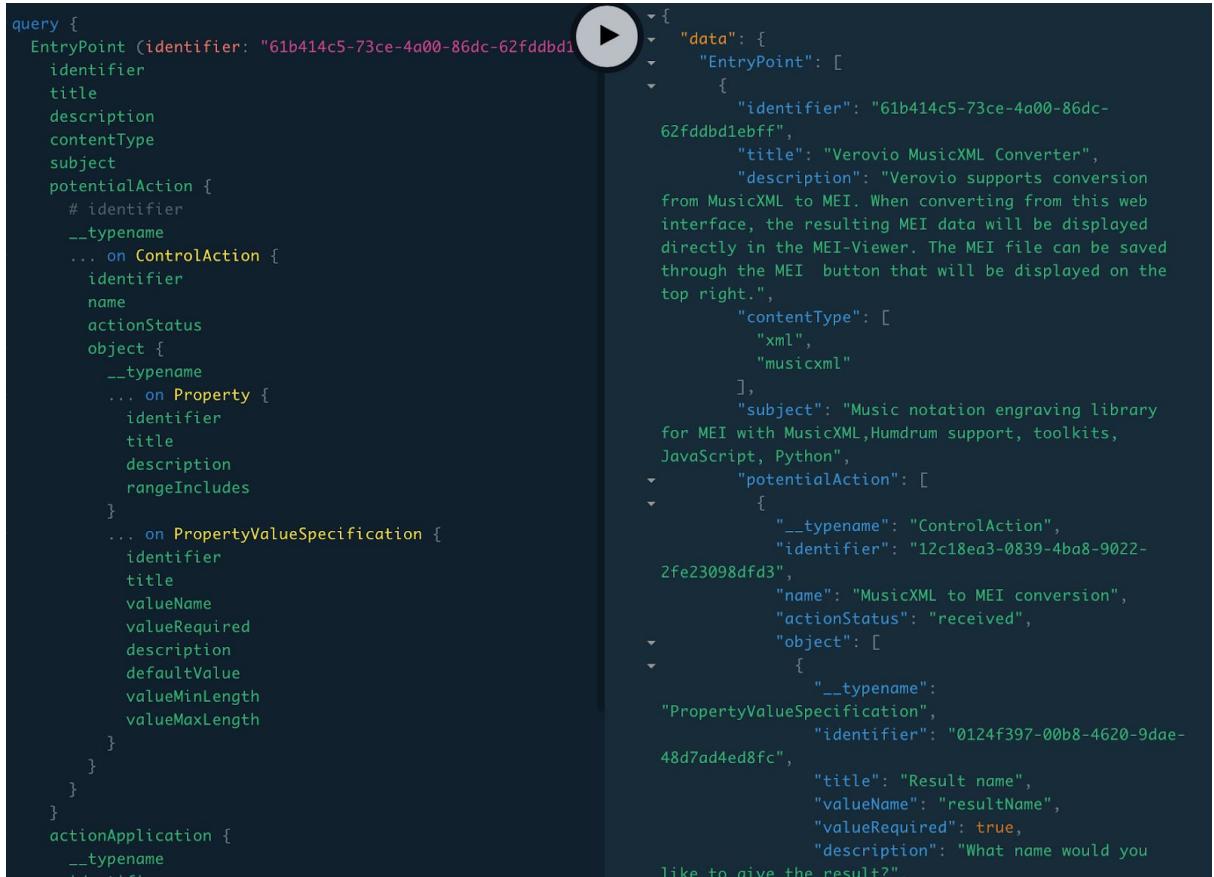
```

mutation {
  AddActionInterfaceThingInterface (
    from: {identifier: "12c18ea3-0839-4ba8-9022-2fe23098fd3"}
    to: {identifier: "0124f397-00b8-4620-9dae-48d7ad4ed8fc", +}
    field: object
  ) {
    from {
      __typename
    }
    to {
      __typename
      ... on PropertyValueSpecification {
        identifier
        title
      }
    }
  }
}

```

Figure 6.13

Query the resulting template model



```

query {
  EntryPoint (identifier: "61b414c5-73ce-4a00-86dc-62fddbd1ebff") {
    identifier
    title
    description
    contentType
    subject
    potentialAction {
      # identifier
      __typename
      ... on ControlAction {
        identifier
        name
        actionStatus
        object {
          __typename
          ... on Property {
            identifier
            title
            description
            rangeIncludes
          }
          ... on PropertyValueSpecification {
            identifier
            title
            valueName
            valueRequired
            description
            defaultValue
            valueMinLength
            valueMaxLength
          }
        }
      }
    }
    actionApplication {
      __typename
    }
  }
}

```

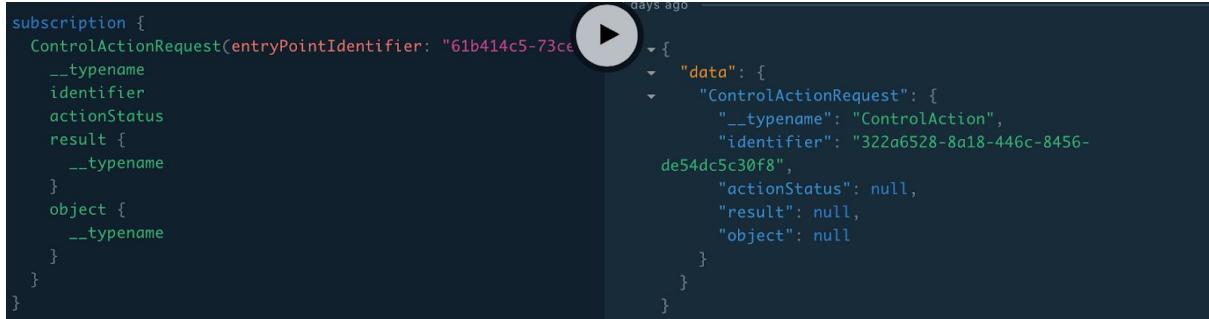
Figure 6.14

Leaving out the identifier from the query will list all available **EntryPoints**, or available algorithm processes that could be offered to Component users.

### 6.3.1.2 Monitor and update instance nodes

This section corresponds to the node data model presented in 6.2.2.

Subscription to RequestControlAction requests, on the basis of the **EntryPoint** identifier



The screenshot shows a GraphQL subscription query on the left and its corresponding JSON response on the right. The query is:

```
subscription {
  ControlActionRequest(entryPointIdentifier: "61b414c5-73ce-4a00-86dc-62fddbd1ebff") {
    __typename
    identifier
    actionStatus
    result {
      __typename
    }
    object {
      __typename
    }
  }
}
```

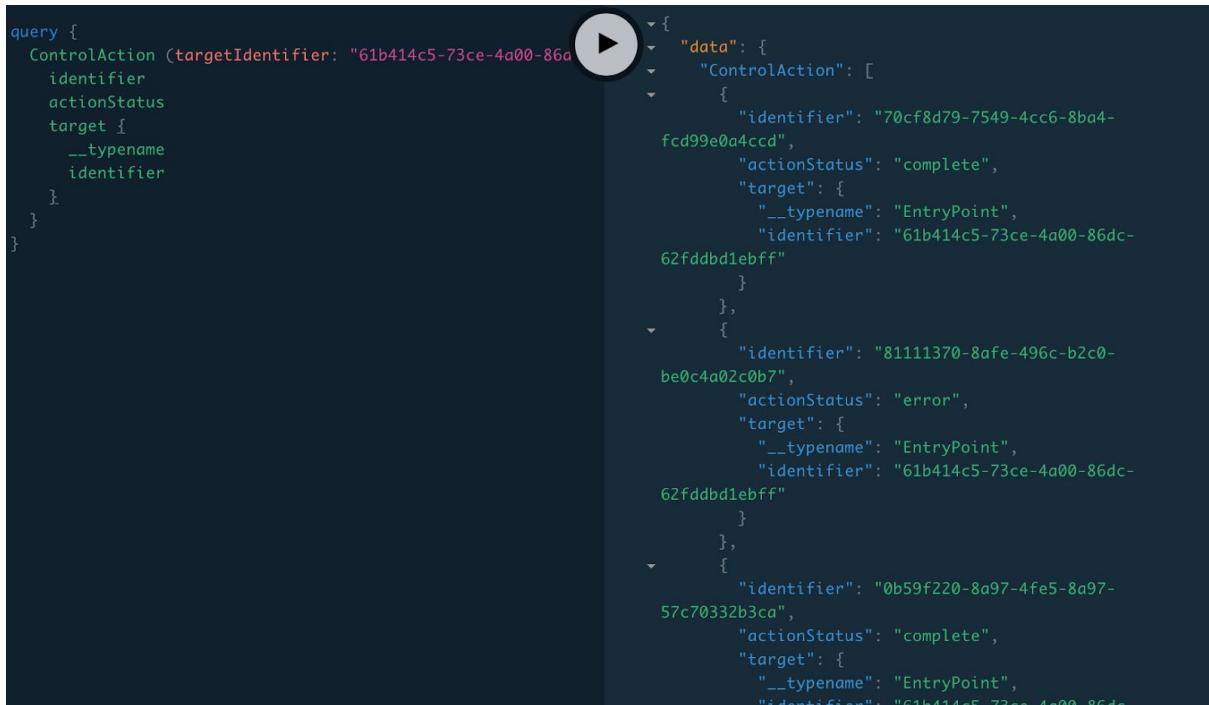
The JSON response shows a single event with a timestamp of 'days ago' and a 'data' field containing a 'ControlActionRequest' object with the specified identifier and other fields.

```
days ago
{
  "data": {
    "ControlActionRequest": {
      "__typename": "ControlAction",
      "identifier": "322a6528-8a18-446c-8456-de54dc5c30f8",
      "actionStatus": null,
      "result": null,
      "object": null
    }
  }
}
```

Figure 6.15

This subscription will set up a websocket connection to the CE api, which will receive a notification of a **ControlAction** being created on the basis of the **EntryPoint** template subscribed to.

It is also possible to query for **ControlActions** created on the basis of a certain **EntryPoint** (*target* property) by adding the targetIdentifier query parameter:



The screenshot shows a GraphQL query on the left and its corresponding JSON response on the right. The query is:

```
query {
  ControlAction(targetIdentifier: "61b414c5-73ce-4a00-86dc-62fddbd1ebff") {
    __typename
    identifier
    actionStatus
    target {
      __typename
      identifier
    }
  }
}
```

The JSON response shows a list of 'ControlAction' objects, each with an identifier, actionStatus, and target (which is an 'EntryPoint' object).

```
{
  "data": [
    {
      "ControlAction": [
        {
          "identifier": "70cf8d79-7549-4cc6-8ba4-fcd99e0a4cc4",
          "actionStatus": "complete",
          "target": {
            "__typename": "EntryPoint",
            "identifier": "61b414c5-73ce-4a00-86dc-62fddbd1ebff"
          }
        },
        {
          "identifier": "81111370-8afe-496c-b2c0-be0c4a02c0b7",
          "actionStatus": "error",
          "target": {
            "__typename": "EntryPoint",
            "identifier": "61b414c5-73ce-4a00-86dc-62fddbd1ebff"
          }
        },
        {
          "identifier": "0b59f220-8a97-4fe5-8a97-57c70332b3ca",
          "actionStatus": "complete",
          "target": {
            "__typename": "EntryPoint",
            "identifier": "61b414c5-73ce-4a00-86dc-62fddbd1ebff"
          }
        }
      ]
    }
  ]
}
```

Figure 6.16

The **ControlAction** identifier thus retrieved can be used to query for the **ControlAction** details:

```

query {
  ControlAction (
    identifier: "d33c8537-c21d-40d6-958d-2425c3fe1691"
  ) {
    identifier
    description
    actionStatus
    object {
      __typename
      ... on PropertyValue {
        name
        valueReference
        value
        nodeValue {
          __typename
          ... on DigitalDocument {
            identifier
          }
        }
      }
    }
    result {
      __typename
    }
  }
}

```

```

{
  "data": {
    "ControlAction": [
      {
        "identifier": "d33c8537-c21d-40d6-958d-2425c3fe1691",
        "description": "MusicXML to MEI conversion",
        "actionStatus": "accepted",
        "object": [
          {
            "__typename": "PropertyValue",
            "name": "MusicXML File",
            "valueReference": "DigitalDocument",
            "value": null,
            "nodeValue": {
              "__typename": "DigitalDocument",
              "identifier": "685152ac-7fd2-4d66-956c-9ec17857f574"
            }
          },
          {
            "__typename": "PropertyValue",
            "name": "resultName",
            "valueReference": "String",
            "value": "asdasd",
            "nodeValue": null
          }
        ],
        "result": null
      }
    ]
  }
}

```

**Figure 6.17**

This template should be set up in such a way that sufficient information can be retrieved from this query to allow the process to be run.

Once received, the algorithm process application could immediately acknowledge the reception by updating the **ControlAction**:

```

mutation {
  UpdateControlAction (
    identifier: "e4a5e889-0d08-4885-acbe-7a023fea69ac"
    actionStatus: received
  ) {
    identifier
    description
    actionStatus
    object {
      __typename
    }
    result {
      __typename
    }
  }
}

```

```

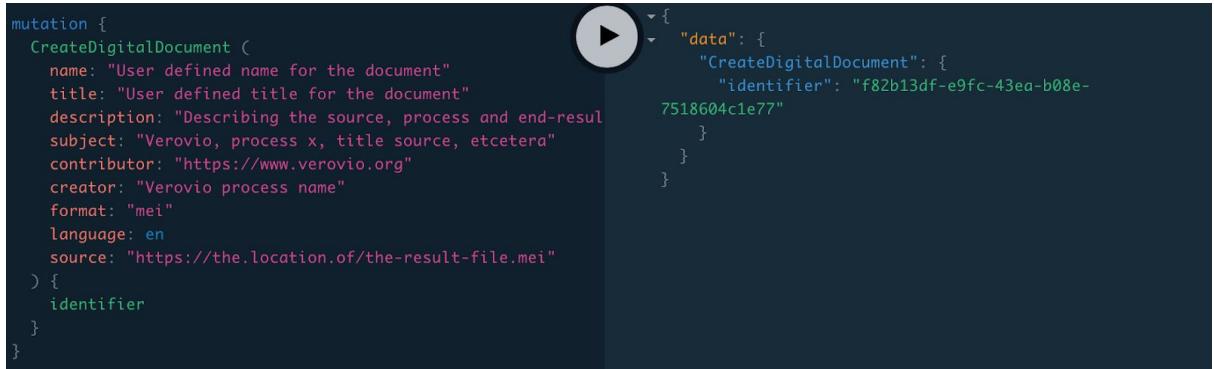
{
  "data": {
    "UpdateControlAction": [
      {
        "identifier": "e4a5e889-0d08-4885-acbe-7a023fea69ac",
        "description": "MusicXML to MEI conversion",
        "actionStatus": "received",
        "object": null,
        "result": null
      }
    ]
  }
}

```

**Figure 6.18**

### 6.3.1.3 Complete the request response cycle

This section corresponds to the node data model presented in 6.2.3. 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:



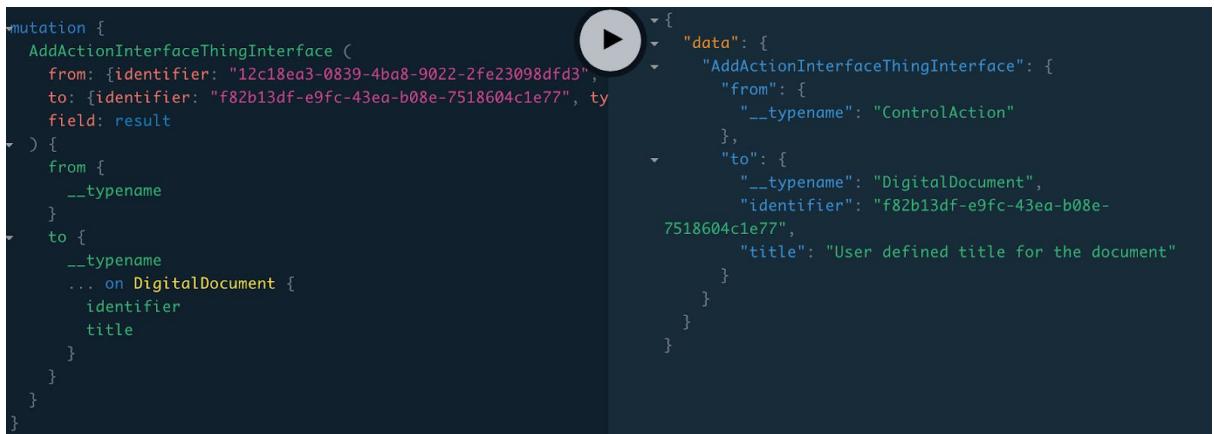
```

mutation {
  CreateDigitalDocument (
    name: "User defined name for the document"
    title: "User defined title for the document"
    description: "Describing the source, process and end-result"
    subject: "Verovio, process x, title source, etcetera"
    contributor: "https://www.verovio.org"
    creator: "Verovio process name"
    format: "mei"
    language: "en"
    source: "https://the.location.of/the-result-file.mei"
  ) {
    identifier
  }
}

```

**Figure 6.19**

With the identifier obtained from the response of the **DigitalDocument** creation, create a *result* relation between **ControlAction** and the produced file:



```

mutation {
  AddActionResultInterfaceThingInterface (
    from: {identifier: "12c18ea3-0839-4ba8-9022-2fe23098dfd3"},
    to: {identifier: "f82b13df-e9fc-43ea-b08e-7518604c1e77", ty: result}
  ) {
    from {
      __typename
    }
    to {
      __typename
      ... on DigitalDocument {
        identifier
        title
      }
    }
  }
}

```

**Figure 6.20**

When the algorithm process application now updates the **ControlAction** `actionStatus`, the process request response cycle will be complete:



```

mutation {
  UpdateControlAction (
    identifier: "e4a5e889-0d08-4885-acbe-7a023fea69ac"
    actionStatus: complete
  ) {
    identifier
    description
    actionStatus
  }
}

```

**Figure 6.21**

## 6.4 Perspective of Component

The goal of WP5 is to create an environment for mid-level integration of components that will be further exploited in WP6 pilots. In order to do so, the data produced in WP3 (musical repertoire, automatic descriptions and generated audio) and annotations delivered through WP4 need to be made accessible and usable in reusable components, meeting common standards.

Component developers can query the CE database for **EntryPoints** that could potentially be interesting for its users. By implementing a user interface on the basis of the information in the (dynamic) template nodes **Property** and **PropertyValueSpecification**, the algorithm process (WP3/4) would become available for a user.

After a user request is sent to the CE API, the Component could set up a subscription to the instantiated **ControlAction** via websocket to any mutations to the created job. The CE would notify the Component of any updates done on the job, most likely by the algorithm process application. It is up to the algorithm process application to determine how fine-grained these updates are.

The Component can show these updates in its UI and act on process completion by making the results available to the user.

With the available result known, the Pilot could create additional relations from this result to other relevant nodes in the CE database, like *isBasedOn* to a **MusicComposition** or *copyrightHolder* to an **Organisation**. This would greatly improve the chances that subsequent users find and re-use the result file.

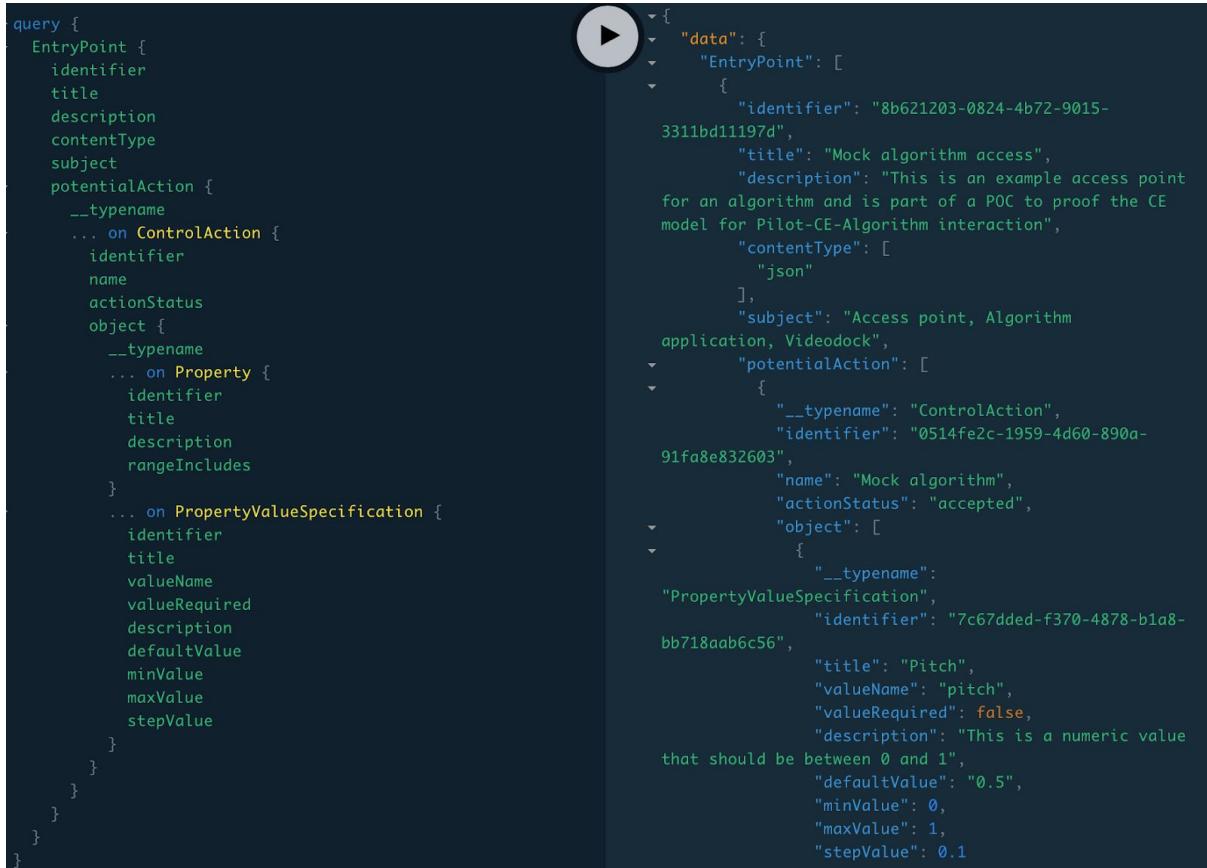
### 6.4.1 GraphQL queries

Following are examples of the GraphQL queries relevant for Component developers. The examples were made using the default [playground](#) environment, which provides rich features for experimenting with and test GraphQL queries. [Apollo](#) offers a number of software libraries that can help integrate GraphQL in an application.

#### 6.3.1.1 Query for available algorithm processes

This section corresponds to the node data model presented in 6.2.1

Query for available EntryPoints:



The screenshot shows a GraphQL playground interface. On the left, there is a code editor containing a GraphQL query. On the right, the results of the query execution are displayed in a tree-view format.

```
query {
  EntryPoint {
    identifier
    title
    description
    contentType
    subject
    potentialAction {
      __typename
      ... on ControlAction {
        identifier
        name
        actionStatus
        object {
          __typename
          ... on Property {
            identifier
            title
            description
            rangeIncludes
          }
          ... on PropertyValueSpecification {
            identifier
            title
            valueName
            valueRequired
            description
            defaultValue
            minValue
            maxValue
            stepValue
          }
        }
      }
    }
  }
}
```

The results pane shows the following JSON data:

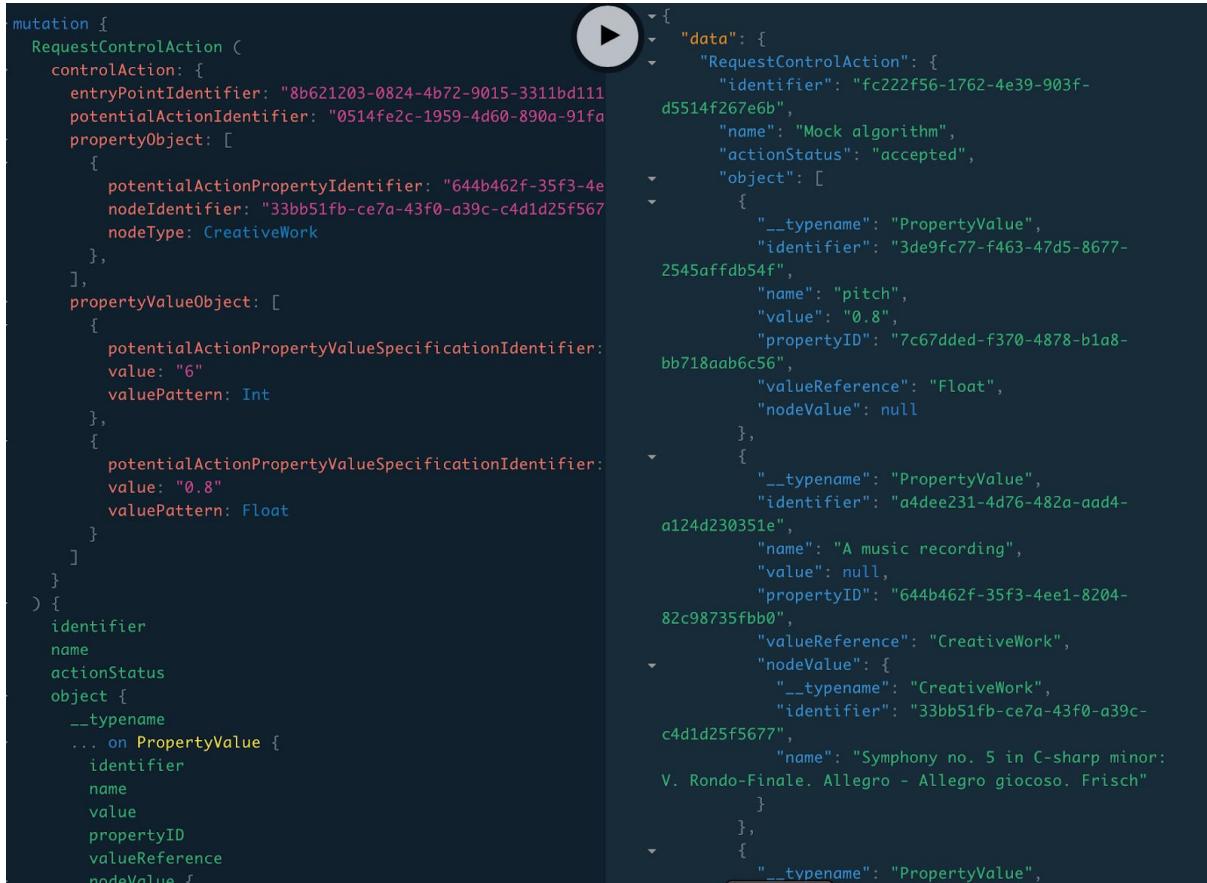
```
{
  "data": {
    "EntryPoint": [
      {
        "identifier": "8b621203-0824-4b72-9015-3311bd11197d",
        "title": "Mock algorithm access",
        "description": "This is an example access point for an algorithm and is part of a POC to proof the CE model for Pilot-CE-Algorithm interaction",
        "contentType": [
          "json"
        ],
        "subject": "Access point, Algorithm application, Videodock",
        "potentialAction": [
          {
            "__typename": "ControlAction",
            "identifier": "0514fe2c-1959-4d60-890a-91fa8e832603",
            "name": "Mock algorithm",
            "actionStatus": "accepted",
            "object": [
              {
                "__typename": "PropertyValueSpecification",
                "identifier": "7c67dded-f370-4878-b1a8-bb718aab6c56",
                "title": "Pitch",
                "valueName": "pitch",
                "valueRequired": false,
                "description": "This is a numeric value that should be between 0 and 1",
                "defaultValue": "0.5",
                "minValue": 0,
                "maxValue": 1,
                "stepValue": 0.1
              }
            ]
          }
        ]
      }
    ]
  }
}
```

Figure 6.22

There should be sufficient information to dynamically create a UI in the frontend.

### 6.3.1.2 Monitor instance nodes

This section corresponds to the node data model presented in 6.2.2. Once a user has chosen a potential job to run, selected the right content and dialed in the parameters, the following request can be made:



The screenshot shows a code editor with two panes. The left pane displays a GraphQL mutation for a 'RequestControlAction' with various fields like 'controlAction', 'propertyObject', and 'propertyValueObject'. The right pane shows the resulting JSON response, which includes a 'data' field containing a 'RequestControlAction' object. This object has fields like 'identifier', 'name' ('Mock algorithm'), 'actionStatus' ('accepted'), and an 'object' array. The 'object' array contains two items, each a 'PropertyValue' object with fields like 'name' ('pitch' or 'A music recording'), 'value' ('0.8' or null), and 'nodeValue' (null or a 'CreativeWork' node). The 'nodeValue' of the second item points to a 'CreativeWork' object with a name like 'Symphony no. 5 in C-sharp minor: V. Rondo-Finale. Allegro. Allegro giocoso. Frisch'.

```

mutation {
  RequestControlAction (
    controlAction: {
      entryPointIdentifier: "8b621203-0824-4b72-9015-3311bd111
      potentialActionIdentifier: "0514fe2c-1959-4d60-890a-91fa
      propertyObject: [
        {
          potentialActionPropertyIdentifier: "644b462f-35f3-4e
          nodeIdentifier: "33bb51fb-ce7a-43f0-a39c-c4d1d25f567
          nodeType: CreativeWork
        },
      ],
      propertyValueObject: [
        {
          potentialActionPropertyValueSpecificationIdentifier:
            value: "6"
            valuePattern: Int
        },
        {
          potentialActionPropertyValueSpecificationIdentifier:
            value: "0.8"
            valuePattern: Float
        }
      ]
    }
  ) {
    identifier
    name
    actionStatus
    object {
      __typename
      ... on PropertyValue {
        identifier
        name
        value
        propertyID
        valueReference
       nodeValue
      }
    }
  }
}

```

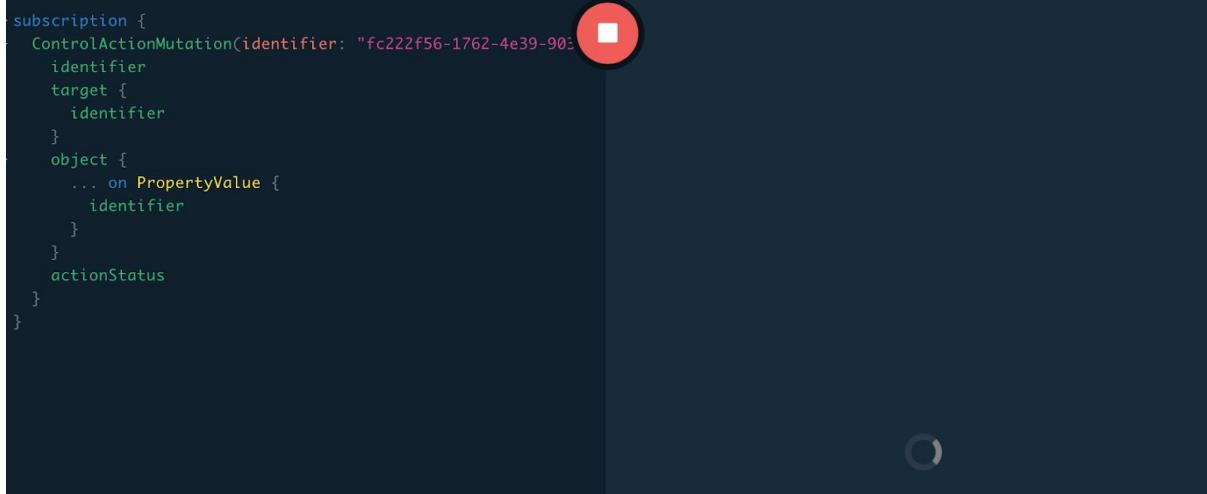
```

{
  "data": {
    "RequestControlAction": {
      "identifier": "fc222f56-1762-4e39-903f-d5514f267e6b",
      "name": "Mock algorithm",
      "actionStatus": "accepted",
      "object": [
        {
          "__typename": "PropertyValue",
          "identifier": "3de9fc77-f463-47d5-8677-2545affdb54f",
          "name": "pitch",
          "value": "0.8",
          "propertyID": "7c67dded-f370-4878-b1a8-bb718aab6c56",
          "valueReference": "Float",
          "nodeValue": null
        },
        {
          "__typename": "PropertyValue",
          "identifier": "a4dee231-4d76-482a-aad4-a124d230351e",
          "name": "A music recording",
          "value": null,
          "propertyID": "644b462f-35f3-4ee1-8204-82c98735fb0",
          "valueReference": "CreativeWork",
          "nodeValue": {
            "__typename": "CreativeWork",
            "identifier": "33bb51fb-ce7a-43f0-a39c-c4d1d25f5677",
            "name": "Symphony no. 5 in C-sharp minor: V. Rondo-Finale. Allegro. Allegro giocoso. Frisch"
          }
        }
      ],
      "__typename": "PropertyValue"
    }
  }
}

```

Figure 6.23

With the returned identifier, the Component can set up a websocket subscription to be informed of any updates on the **ControlAction** ‘job’:



The screenshot shows a code editor with a single pane containing a websocket subscription query. It uses the same 'ControlActionMutation' from Figure 6.23 and adds a 'target' field to receive notifications when the 'ControlAction' is updated.

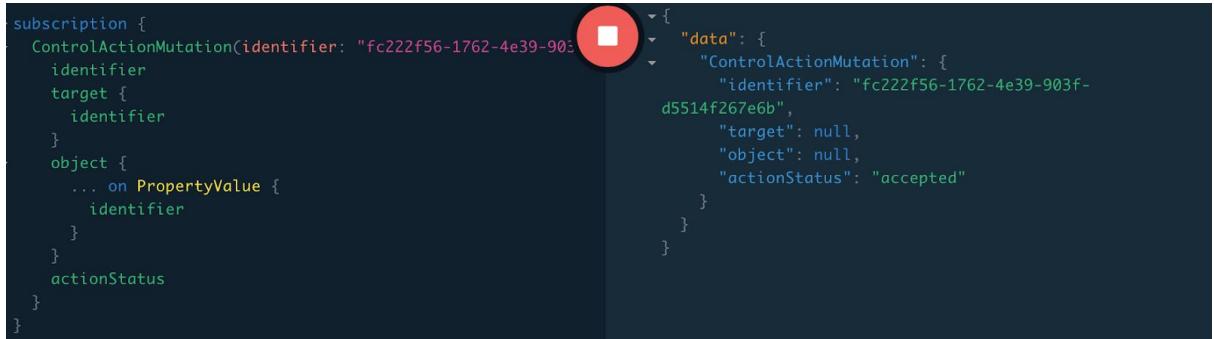
```

subscription {
  ControlActionMutation(identifier: "fc222f56-1762-4e39-903f-d5514f267e6b") {
    target {
      identifier
    }
    object {
      ... on PropertyValue {
        identifier
      }
    }
    actionStatus
  }
}

```

Figure 6.24

Every time this **ControlAction** gets updated, the Component will receive a notification, and the Component UI can be updated accordingly:



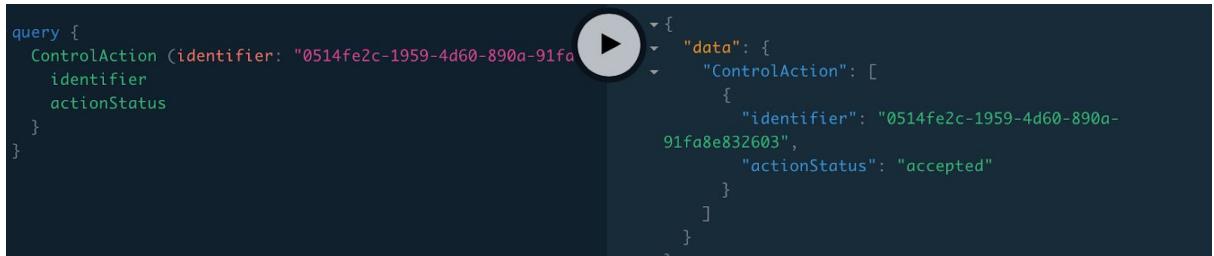
```

subscription {
  ControlActionMutation(identifier: "fc222f56-1762-4e39-903f-d5514f267e6b") {
    identifier
    target {
      identifier
    }
    object {
      ... on PropertyValue {
        identifier
      }
    }
    actionStatus
  }
}

```

Figure 6.25

It is of course also possible to regularly poll the **ControlAction** for any changes:



```

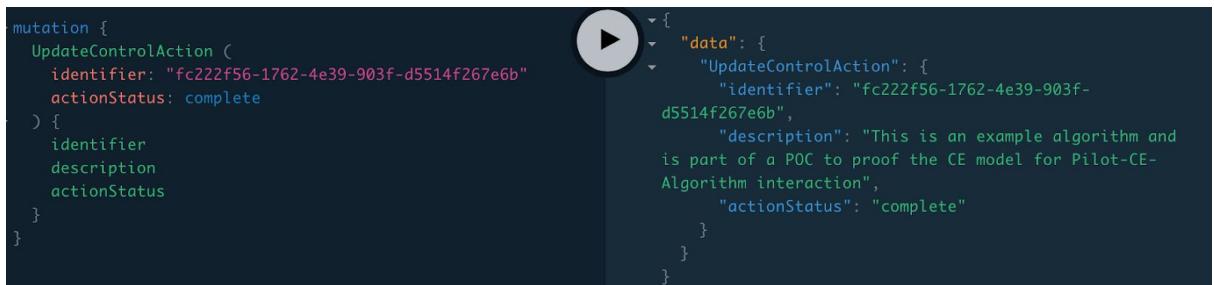
query {
  ControlAction (identifier: "0514fe2c-1959-4d60-890a-91fa8e832603") {
    identifier
    actionStatus
  }
}

```

Figure 6.26

#### 6.4.1.3 Complete the request response cycle

This section corresponds to the node data model presented in 6.2.3. Once the **ControlAction** status has notified it is ‘complete’:



```

mutation {
  UpdateControlAction (
    identifier: "fc222f56-1762-4e39-903f-d5514f267e6b"
    actionStatus: complete
  ) {
    identifier
    description
    actionStatus
  }
}

```

Figure 6.27

The Component can then fetch the URL of the result (*source* property) by querying the **ControlAction.result** property:

```

query {
  ControlAction (identifier: "12c18ea3-0839-4ba8-9022-2fe23098dfd3") {
    identifier
    name
    actionStatus
    result {
      __typename
      ... on DigitalDocument {
        name
        source
      }
    }
  }
}

{
  "data": {
    "ControlAction": [
      {
        "identifier": "12c18ea3-0839-4ba8-9022-2fe23098dfd3",
        "name": "MusicXML to MEI conversion",
        "actionStatus": "received",
        "result": {
          "__typename": "DigitalDocument",
          "name": "User defined name for the document",
          "source": "https://the.location.of/the-result-file.mei"
        }
      }
    ]
  }
}

```

**Figure 6.28**

And make it available to the user.

## 6.5 Perspective of CE

The role of the CE in this mechanism is to maintain the data model and custom mutations that will enable Component and process algorithm application developers to create and follow **ControlActions** that effectively behave like jobs. This model should allow Component-WP3/4 interactions to take place as frictionless as possible, yet assuring the CE retains the position of middleman for all these interactions, as this is what ensures that user-interactions and process results lead to meaningful contributions the larger TROMPA dataset.

If the current model or custom functionalities present limitations, CE developers should consider to fix or extend CE api code in consultation with participants at the earliest opportunity. Backwards compatibility should be maintained with an effective versioning strategy.

## 7. Integration of frontend components

One of the core ideas of the project is that generic components can be reused in different pilots and end-user applications. At this point, we distinguish the following frontend (as in browser application) components that could and should be reused in the first release of pilot applications in M24.

- ❖ CE Multimodal component<sup>3</sup>. This is a React library that can be re-used in a React Javascript project to have easy access to common search queries to the CE and visualisation of results of objects stored in the CE, currently implemented (v0.1.0) are searches for works, persons and scores. Additional search features can be added as pilots require them. The multimodal component will become available through package manager NPM (public archive) and Yarn. The component can be used as an overlay or inline to search for items in the CE-API. When an item is clicked, the component will notify this to its parent component using a callback prop with the details of the item. The component allows developers to develop custom facets and filters. The custom facet uses a separate GraphQL query or a fixed list of possible options. When a user enables a facet, the selected option will be given to the GraphQL

---

<sup>3</sup> <https://github.com/trompamusic/ce-multimodal-component>

function. This makes it possible to use the active facets in the final search query. There are four possible levels of integration of the CE Multimodal component:

- Access to a set of common GraphQL queries that can be configured using parameters.
- Adjust the query that is being used in the CE Multimodal component.
- Reuse a visual component to provide an input for users and perform these searches.
- Change the appearance of the CE Multimodal component search results.
- ❖ CE Digital Score Edition component<sup>4</sup>. This is a React library that can be used to render MEI scores as SVGs within a web client, and supports the creation and viewing of annotations upon the score.

Other components might be defined during the project. We agree that the frontend components or libraries that could be of use to TROMPA partners should comply with the following technical requirements:

- ❖ The React component can be used with the latest version of React.
- ❖ The URL of the GraphQL endpoint targeted by the CE-API should configurable without the need to compile from source.
- ❖ The React component will accept props to control the behavior of the component.
- ❖ The React component can be used as a Controlled Component<sup>5</sup>.
- ❖ The React component can be styled using a ThemeProvider which supports overwrites using JSS.

## 8. Conclusion

In this deliverable we specified the strategies for technical integration of data generated by different technologies. The document is written for a technical audience and should have provided practical information and guidelines for developers and researchers on how to integrate with the Contributor Environment (CE). It is expected that the document is further updated and improved after its initial submission.

The main contents of the document started (section 2) with a detailed overview of the internal data model of the TROMPA Contributor Environment (CE). In section 3 it described best practices for setting properties and relations when managing data in the CE in the form of guidelines. In sections 4 and 5 the interfaces for interacting with the CE were documented. In chapter 6 it was explained how the job workflows and processes that are developed in WP3 and WP4 can be integrated with the CE. Section 7 provided requirements for the frontend components that can be reused in different end-user pilots.

The deliverable provides all the information needed for the partners of TROMPA to develop their integration with the CE and develop the first prototypes of the pilots.

---

<sup>4</sup> <https://github.com/trompamusic/DigitalScoreEdition>

<sup>5</sup> <https://reactjs.org/docs/forms.html#controlled-components>

## 9. References

### 9.1 List of abbreviations

Use the following table format

Abbreviation	Description
UPF	University Pompeu Fabra
TUD	Technische Universiteit Delft
GOLD	Goldsmiths College
MDW	University of Music and Performing Arts Vienna
VD	Video Dock BV
PN	Peachnote GmbH
VL	Vocstro Labs, S.L.
RCO	The Royal Concertgebouw Orchestra
CDR	Stichting Centrale Discotheek Rotterdam
MCM	Music Connection Machine
IMSLP	Petrucci Music Library (IMSLP.org)
MOOC	Massive Online Open Course
MEI	Music Encoding Initiative
RDF	Resource Description Framework
API	Application Programming Interface
GDPR	General Data Protection Regulation