TROMPA

TROMPA: Towards Richer Online Music Public-domain Archives

Deliverable 2.3

Technical Requirements and Integration v2

Grant Agreement nr	770376
Project runtime	May 2018 - April 2021
Document Reference	TR-D2.3-Technical Requirements and Integration v2
Work Package	WP2 - Project Coordination
Deliverable Type	Report
Dissemination Level	PU
Document due date	31 March 2021
Date of submission	31 March 2021
Leader	5 - VD
Contact Person	Bauke Freiburg (bauke@videodock.com)
Authors	Wim Klerkx (VD), Bauke Freiburg (VD), Christiaan Scheermeijer (VD), Roy Schut (VD), Alastair Porter (UPF), David Weigl (MDW)
Reviewers	Aggelos Gkiokas (UPF)

Executive Summary

The first version of this deliverable started out as the technical requirements and integration report for the TROMPA project and specified the strategies for technical integration of data, technologies and end-user applications. The initial version set the integration guidelines to be followed during the project. An intermediate version was made publicly available outside the TROMPA consortium at M30 as a reference companion to the D5.1-2 - Data Infrastructure deliverable. This third version has been updated at the M35 milestone to reflect the latest and final version of the TROMPA Contributor Environment.

This document is written for a technical audience and provides practical information and guidelines for developers and researchers on how to integrate with the Contributor Environment (CE). The objective of this document is that third parties with little background information on the project understand how they can integrate with the CE, based on this technical documentation.

First, in the introduction, naming- and style conventions are addressed.

Section 2 presents the internal data model of the CE which 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 are 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 are 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.

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.

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 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. 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 only contain information on the thing the web resource is about. All metadata properties are 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, multiple **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**. 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. Detailed examples for **queries**, **mutations** and **subscriptions** are provided in this deliverable.

As described in selection 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. A subscription mechanism can enable both the Component and Process system to be actively updated on job creation and status updates in real time. 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 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 is 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 regularly querying the CE database for new instances of template nodes (**ControlActions).** After a new request comes 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 becomes 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 **ControlAction**s that effectively behave like jobs. This model allows interactions to take place as frictionless as possible, yet assuring the CE retains the position of middleman for all these interactions.

Version Log		
#	Date	Description
v0.1	28 September 2020	Initial version based on D2.3 deliverable
v1.0	31 March 2021	Version ready for publication

Table of Contents

1. Introduction

The first version of this deliverable started out as the technical requirements and integration report for the TROMPA project and specified the strategies for technical integration of data, technologies and end-user applications. The initial version set the integration guidelines to be followed during the project. An intermediate version was made publicly available outside the TROMPA consortium at M30 as a reference companion to the 2nd version of deliverable **D5.1 - Data Infrastructure**[1]. This third version has been updated at the M35 milestone to reflect the latest and final version of the TROMPA Contributor Environment.

This document is written for a technical audience and provides practical information and guidelines for developers and researchers on how to integrate with the Contributor Environment (CE). The objective of this document is that third parties with little background information on the project understand how they can integrate with the CE, based on this technical documentation.

The rest of this introductory section provides naming and style conventions for the document. The main contents of the document starts (section 2) with a detailed overview of the internal data model of the TROMPA Contributor Environment (CE). Section 3 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 W3C Schema.org Community Group (Schema.org)[2] structured data vocabulary. These functionalities can be accessed through a GraphQL and a RESTful API interface. In section 6 it is explained how the job workflows and processes that are developed in WP3 and WP4 can be integrated with the CE.

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)

[1] https://trompamusic.eu/deliverables/TR-D5.1-Data_Infrastructure_v2.pdf
[2] https://schema.org/

2. Internal data model

This section presents the internal data model of the CE which 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 CE 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**[3]. **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, it is further extended with types like **AudioObject** and **VideoObject**.

[3] https://schema.org/Thing

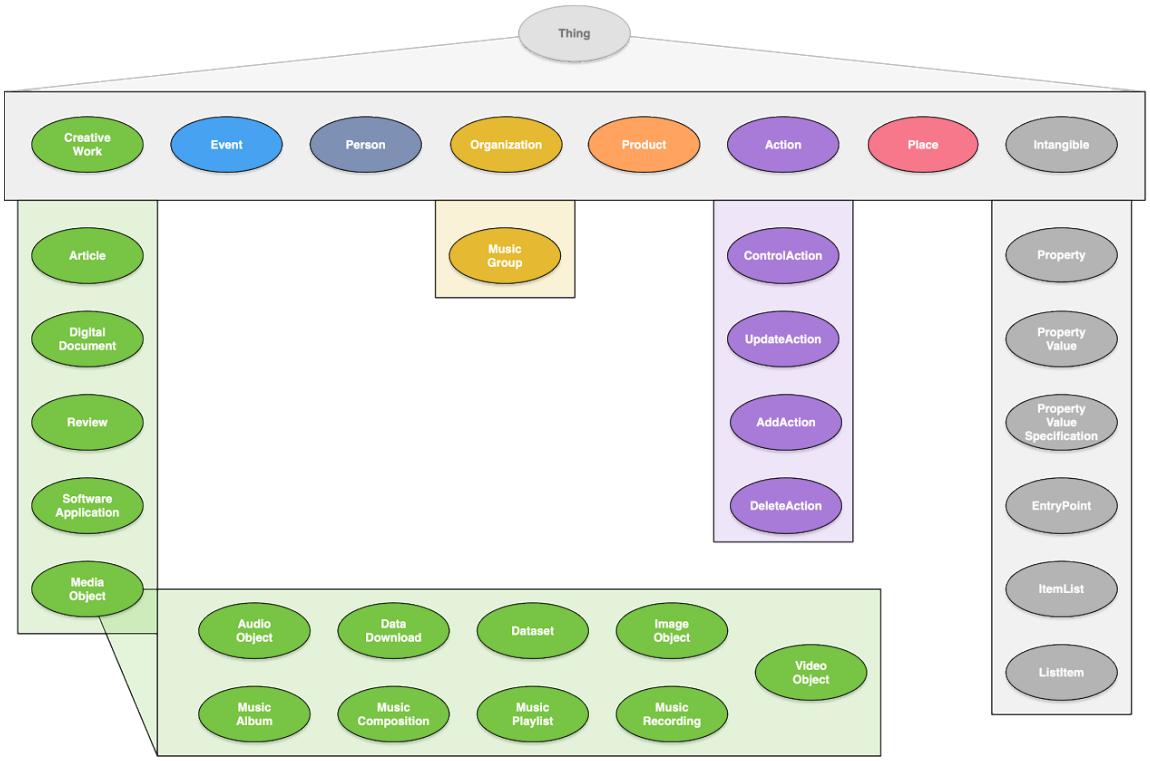

Figure 2.1. Schema.org base types, plus TROMPA relevant extensions

The following types are 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**
  - **DigitalDocument**
  - **MediaObject**
    - **AudioObject**
    - **Dataset**
    - **MusicComposition**
    - **MusicPlaylist**
    - **MusicRecording**
  - **SoftwareApplication**
  - **VideoObject**
- **Event**

- **Intangible**
  - **DefinedTerm**
  - DefinedTermSet
  - **Property**
  - **PropertyValue**
  - **PropertyValueSpecification**
  - **EntryPoint**
  - **ItemList**
  - **ListItem**
  - **Rating**
- **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[4] we set out using the 15 classic metadata terms (DCMES)[5], 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*

[4] https://www.dublincore.org/
[5] http://dublincore.org/documents/dces/

Early 2020, after some practical experience with this model, we ran into the limitations of the 15 DCMI elements; e.g. there exists a way to capture a generic "date", but not to differentiate between a "date created", a "date submitted", or a "date modified", which was emerging as a requirement.

Given that the 15 elements are mirrored in the terms namespace, a wholesale shift to DCMI terms[6] from DCMI elements was implemented. The most useful properties and classes of DCMI Metadata Terms are published as ISO 15836-2:2019[7]

2.1.3 SKOS equivalence linking properties

From the Simple Knowledge Organization System[8] (SKOS) ontology, five of the six mapping properties [9] are adopted to allow equivalence interlinking.

The CE contains 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 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 or a super class. 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. Language is used to describe the language that the metadata of the node is written in. If the item described by a node is written in a known language (e.g. lyrics), inLanguage is used.

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[10] were adopted on the relevant types:

[6] https://dublincore.org/specifications/dublin-core/dcmi-terms/#section-1
[7] https://www.iso.org/obp/ui/#iso:std:iso:15836:-2:ed-1:v1:en
[8] https://www.w3.org/TR/skos-primer
[9] https://www.w3.org/TR/skos-reference/#mapping
[10] https://www.w3.org/TR/2013/NOTE-prov-primer-20130430/

- *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

In the case that an agent needs to store additional properties on a node, we provide a custom *additionalProperty* property[11]. 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.

[11] Type https://vocab.trompamusic.eu/vocab#AdditionalProperty

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 overwhelmingly 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 collected 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.

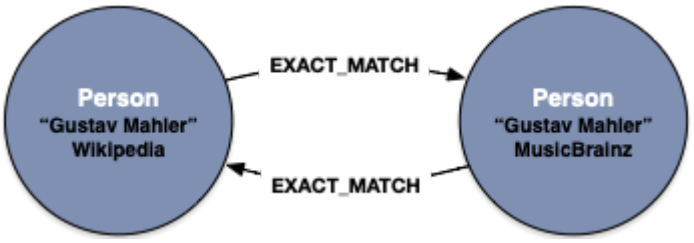

Figure 3.1 multiple nodes representing complementary information about the same entity

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; for example an alignment file produced by a participant will be stored at an external URL and can be accessed through the CE as a

DigitalDocument 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.

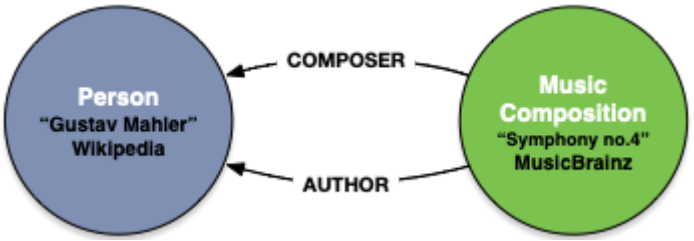

Figure 3.2 unidirectional relations

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

Figure 3.3 multiple relations with the same property name

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.

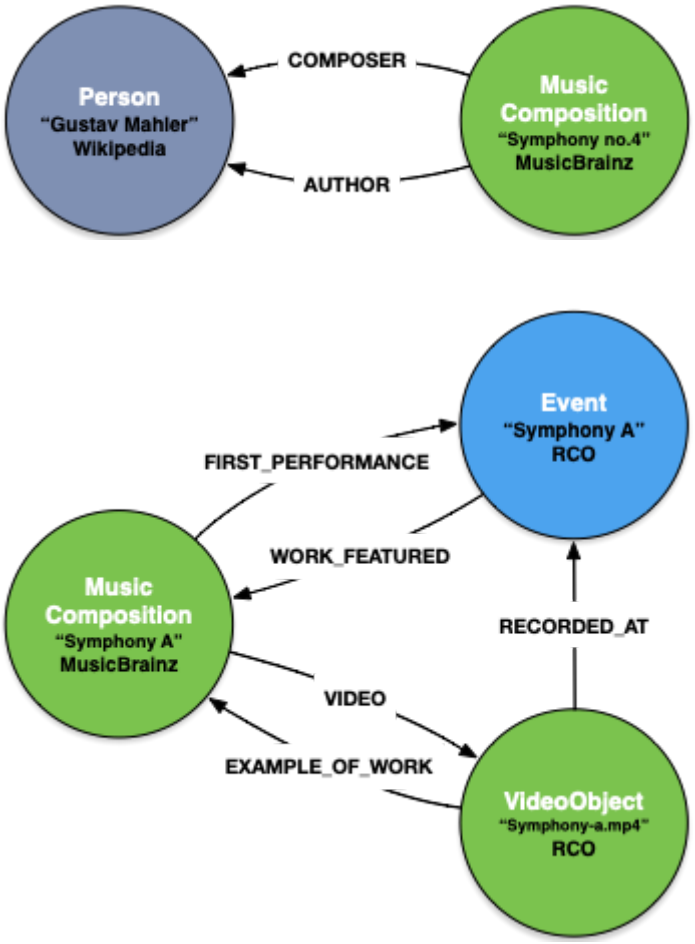

Figure 3.4 multiple relation types between nodes

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.

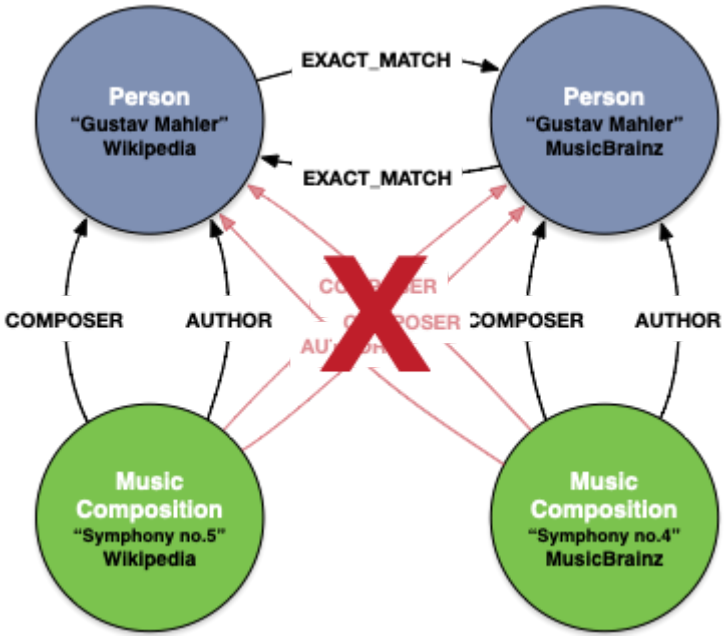

Figure 3.5 unnecessary to duplicate the relations to the exact matching nodes

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 available in Appendix A.

3.2.3. Equivalence linking properties

Most interlinking between nodes stored in the CE is done through the default properties provided by the schema.org base types and the extensions described in the previous chapter.

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.

From the SKOS we adopt the mapping ontology that allows to define several levels of equivalence:

Property	Value type	Explanation/example
exactMatch	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.' *exactMatch* 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 ...*Match* properties should be set. Two web resources about the same composer should each have the *exactMatch* property set with the other.
closeMatch	Parent type	'used to link two concepts that are sufficiently similar that they can be used interchangeably in **some** information retrieval applications' *closeMatch* 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 ...*Match* properties should be set. Two score versions of the same music composition should both have the *closeMatch* property set with the other.
broadMatch *narrowMatch*	Parent type	'used to state a hierarchical mapping link between two concepts.' *broadMatch* and *narrowMatch* are inverses; when setting either in one node, the inverse should be set in the related node. If either is set, none of the other ...*Match* properties should be set. A node for full score can have the *narrowMatch* property set with a node representing one page of the same score. The one page node could then set the *broadMatch* property with another.
relatedMatch	**ThingInterface** (any base and extended type)	'used to state an associative mapping link between two concepts.' If set, none of the other ...*Match* properties should be set. *relatedMatch* 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)

		A music group that operates under different names and/or occupancies can have different nodes that are interrelated by *relatedMatch* relations. Should not be set when any of the other ...Match properties are set.

Table 3.2. Equivalence and examples

3.2.4 Internationalization properties

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 when using the REST API will yield language filtered results. 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.

The `Accept-Language` header can contain multiple language codes. If a node is available in any of the specified languages, it will be returned. Setting the `Accept-Language` to 'fr,en' would yield a result if the language of the node is set to either fr or en.

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. If an item doesn't point to an existing node in the CE, the **ListItem**.*name* field can be used to describe the item.
- 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.

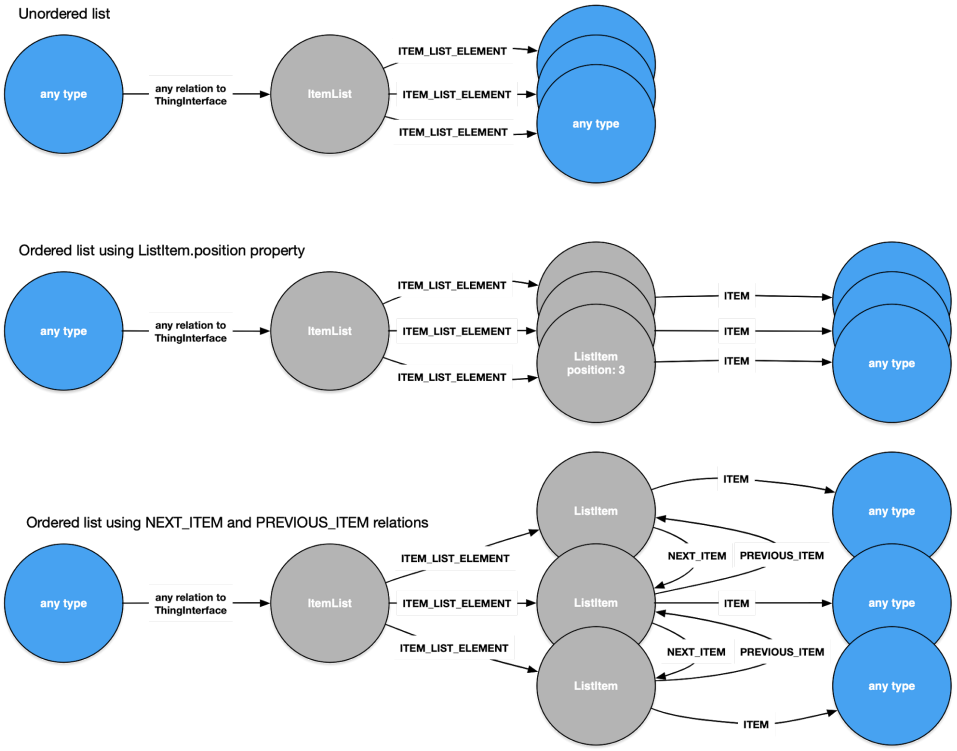

Figure 3.6 Item lists schema (unorder, ordered)

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. This allows the creation of metadata tracking the provenance of nodes created in the CE, e.g., during data import or in response to outcomes of algorithms orchestrated by the TROMPA Processing Library. Figure 3.X illustrates this idea: Using the interface for integration of software agents with the CE (Section 6), the TROMPA Processing Library responds to a request to execute the Verovio software agent. The agent processes a MusicXML-encoded digital score, stored on the Web and referenced by a CE node using the contentUrl property, and generates a new, MEI-encoded equivalent score, with a corresponding new node. Provenance interactions tracking this process are captured and stored in the CE using PROV-O properties.

As this is a rich subject, please read the W3C PROV-O primer[12], while keeping in mind that only the Starting Point Terms[13] are implemented in the CE.

[12] https://www.w3.org/TR/2013/NOTE-prov-primer-20130430/
[13] https://www.w3.org/TR/2013/REC-prov-o-20130430/#description-starting-point-terms

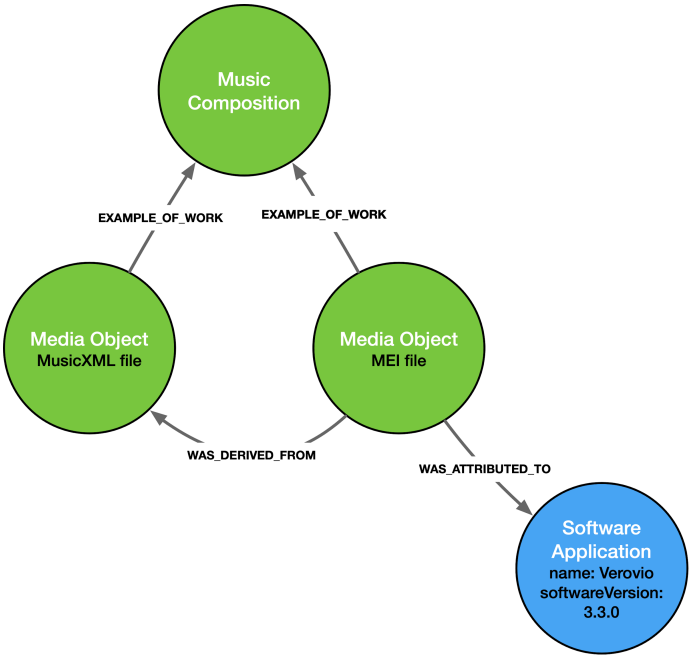

Figure 3.7 Provenance properties

3.2.7. Additional properties

An additional property can be added to any single node by adding a node of type **PropertyValue**[14] 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 **Property.***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.

In the following table we disambiguate some properties that have partial or complete overlapping semantics:

[14] https://meta.schema.org/PropertyValue

Ontology	Property	Value type	Description
Schema.**Thing**	*sameAs*	URL	Equal to DC.*source*
Schema.**Thing**	*url*	URL	URL of the webresource about the entity. In case of a Person, this is equal to Schema.**Thing**.*sameAs* and DC.*source*. In case of a content file, this is equal to Schema.**CreativeWork.***contentUrl*
Schema.**CreativeWork**	*contentUrl*	URL	URL of the image, audio or video file
DC	*source*	URL	Equal to Schema.**Thing**.*sameAs*
DC	*creator*	String/URL	For types where this makes sense (**CreativeWork**, **Action**, **Event**) the value is a URL representing the agent who created the node, either a link to the source code of a software tool, or the WebID of a Solid user.
DC	*publisher*	URL	For **CreativeWork**, this corresponds to the base-URL of the *schema_publisher*, which is the **Organization** that published the book, article, score etc.
DC	*contributor*	URL	For all types, this value is the base-URL of the web-resource about the thing, if available. This can equal publisher.
Schema.**MusicComposition**	*composer*	Person	For **MusicComposition** this value is the same as *author.*

Schema.**CreativeWork**	*author*	Person	For **MusicComposition** this value is the same as *composer.*
Schema.**Person**	*birthDate*	Date	Is equal to DC.*date*
Schema.**Event**	*startDate*	DateTime	Is equal to DC.*date*
DC	*date*	Date	The value represents the date for the thing coming into existence. E.g. the publication date (**Article**), first performance date (**MusicComposition**). In case of a **Person**, it is equal to the *birthdate.* In case of an **Event**, it is equal to *startDate*
Schema.**Thing**	*inLanguage*	String	On **CreativeWork**, **Event** and **Action** 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. This can be the same, or can contain the same value as DC.*language* This value should be left empty if **CreativeWork** and **Event** cannot be determined.
DC	*language*	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

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. 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 URI's in the additionalType property available for each internal model type.

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

3.5 Ontological guidelines

In order to ensure consistency in the data added to the CE by each partner, we provide concrete guidelines about which fields to set on the object types that we are currently using. This documentation[15] is available online and will be maintained beyond the publication of this document.

3.6 Private data guidelines

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 are 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. 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 by the user) 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).

[15] https://trompace-client.readthedocs.io/en/latest/

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).

Solid[16] a Web decentralisation project building on a W3C standards-based Linked Data technology stack, is the primary implementation mechanism providing user-controlled external storage spaces and decentralised authorisation and authentication mechanisms.

Solid aims to enable rich online interactions between users that retain data ownership with each contributing 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 (Personal Online Datastores) 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 by UPF at https://trompa-solid.upf.edu/ for 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.

[16] http://solidproject.org

4. GraphQL interface for managing data

GraphQL[17] 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[18] can provide detailed background information for the following sections.

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

- Queries, for retrieving data
- Mutations, for adding or updating data
- Subscriptions, for receiving notifications of changes to data in real-time

For the CE-API, a graphical interface[19] is available that provides a human-friendly way to interact with the GraphQL API interface and supports rich introspection of the schema.

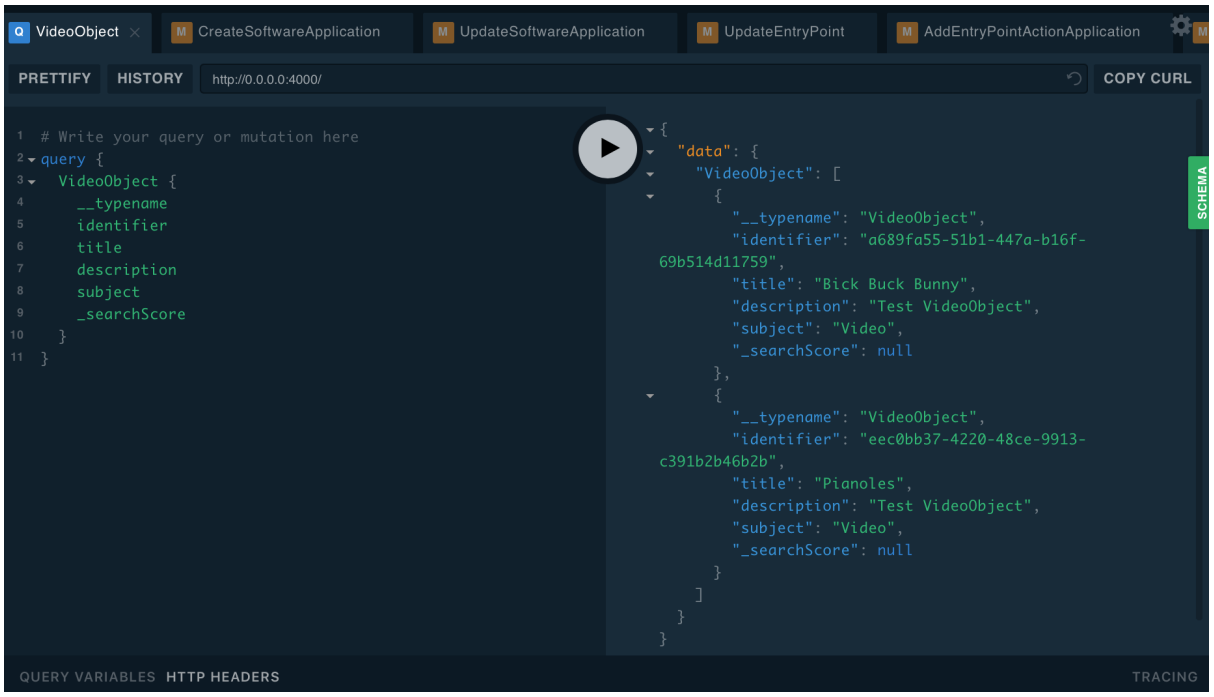

Figure 4.1. The top part of the interface is for query management. The left part is used to help composing a request and the right part will show the response after clicking the **>** button.

[17] https://graphql.org/
[18] https://graphql.org/learn/
[19] https://api-test.trompamusic.eu/

4.1 Schema introspection

On the right-hand side of the interface's query editor, 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.

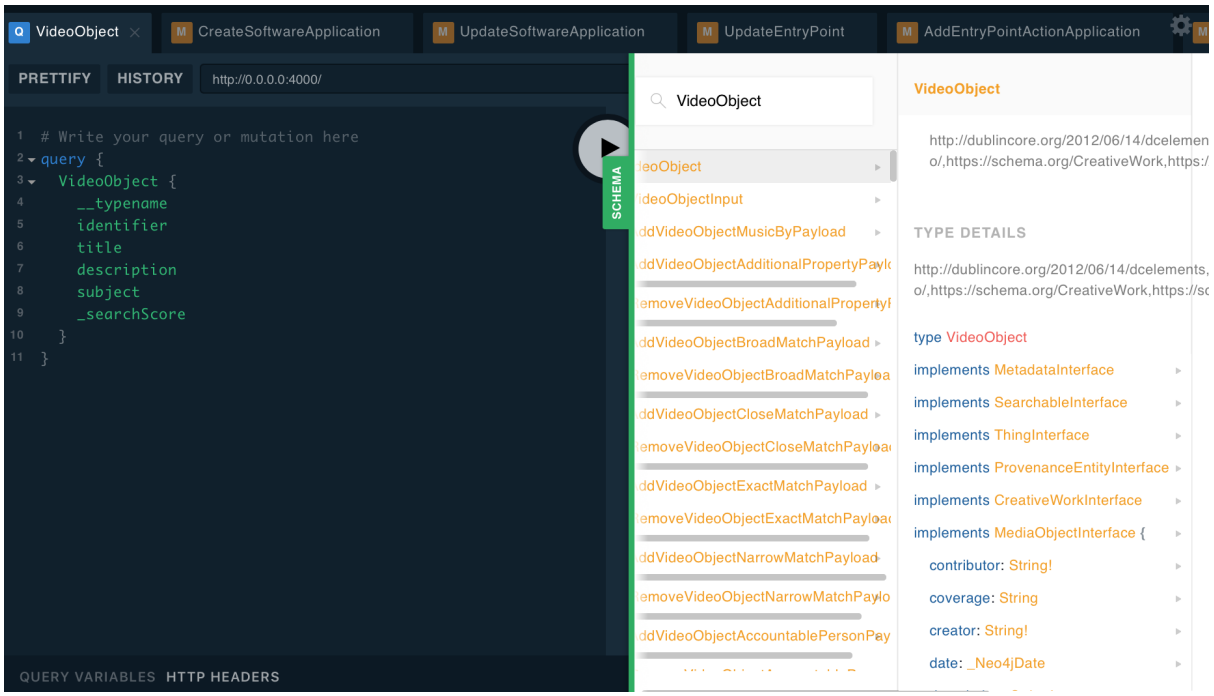

Figure 4.2. This introspection tool shows the actual schema and its possibilities, and is the best way to explore the schema and how to access the CE-API functionalities.

4.2. Queries

Queries are requests for existing data from the database.

4.2.1 Simple query for one node

Query:

```
query {
 Person (
    identifier: "349c6350-3945-4d36-ae88-e1b763f3d432"
  ) {
    identifier
    name
    source
    title
    birthDate {
      formatted
    }
    birthPlace {
      identifier
      name
    }
  }
}
```

Response:

```
{
  "data": {
    "Person": [
      {
        "identifier": "349c6350-3945-4d36-ae88-e1b763f3d432",
        "name": "Ludwig van Beethoven",
        "source": "https://musicbrainz.org/artist/1f9df192-a621-4f54-8850-2c5373b7eac9",
        "title": "Ludwig van Beethoven - MusicBrainz",
        "birthDate": {
          "formatted": "1770-12-17"
        },
        "birthPlace": {
          "identifier": "668218cf-a1fc-46f2-81a0-e27ae074468e",
          "name": "Bonn"
        }
      }
    ]
  }
}
```

Figure 4.3 Simple query example

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

Query:

```
query {
 MusicComposition(first: 3, offset: 1) {
    identifier
    name
  }
}
```

Response:

```
{
  "data": {
    "MusicComposition": [
      {
        "identifier": "33952216-e3fe-4877-8dd8-c59b0671ee02",
        "name": "Das klagende Lied: I. Waldmärchen"
      },
      {
        "identifier": "9a9e7e88-4f94-46d5-895f-c6bce80fdc50",
        "name": "Das klagende Lied: II. Der Spielmann"
      },
      {
        "identifier": "f4d6c361-e21f-4949-b466-de10a9f35cf5",
        "name": "A bente y siete de março"
      }
    ]
  }
}
```

Figure 4.4

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

4.2.3 Complex query

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. The CE-api interface suggests options and validates the query against the schema in real time.

Query:

```
query {
 MusicComposition(identifier: "9a9e7e88-4f94-46d5-895f-c6bce80fdc50") {
    identifier
    name
    datePublished {
      formatted
    }
    isPartOf {
      ... on MusicComposition {
        identifier
        name
        composer(first: 1) {
          ... on Person {
            identifier
            name
          }
        }
      }
    }
    composer(first: 1) {
      ... on Person {
        identifier
        name
      }
    }
  }
}
```

Response:

```
{
  "data": {
    "MusicComposition": [
      {
        "identifier": "9a9e7e88-4f94-46d5-895f-c6bce80fdc50",
        "name": "Das klagende Lied: II. Der Spielmann",
        "datePublished": {
          "formatted": null
        },
        "firstPerformance": null,
        "isPartOf": {
          "identifier": "660949b4-653e-4841-aa3b-262e9ed1c5fe",
          "name": "Das klagende Lied",
          "composer": [
            {
              "identifier": "67a66f99-98cd-471e-b851-4db6fffb8f29",
              "name": "Gustav Mahler"
            }
          ]
        },
        "composer": [
          {
            "identifier": "67a66f99-98cd-471e-b851-4db6fffb8f29",
            "name": "Gustav Mahler"
          }
        ]
      }
    ]
  }
```

```
}
```

Figure 4.5 Complex query example

4.3. Mutations

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

4.3.1 Creating a node

Query:

```
mutation {
 CreateMusicComposition(
    creator: "https://github.com/trompamusic/trompa-ce-client/tree/master/demo"
    contributor: "https://musicbrainz.org"
    source: "https://musicbrainz.org/work/5b8cb51a-6b4f-48b7-888d-c7f1e812dfba"
    format: "text/html"
    title: "Symphony no. 1 in D major \"Titan\" - MusicBrainz"
    name: "Symphony no. 1 in D major \"Titan\"```
    language: en
  ) {
    identifier
    name
    title
    creator
    contributor
  }
}
```

Response:

```
{
  "data": {
    "CreateMusicComposition": {
      "identifier": "f7f4175a-714d-4b6f-81c1-c2bb8e52c1e4",
      "name": "Symphony no. 1 in D major \"Titan\"",
      "title": "Symphony no. 1 in D major \"Titan\" - MusicBrainz",
      "creator": "https://github.com/trompamusic/trompa-ce-client/tree/master/demo",
      "contributor": "https://musicbrainz.org"
    }
  }
}
```

Figure 4.6 Node creation example

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

Relationships between nodes are created by a separate join mutation, rather than being provided in the initial mutation(described below).. The Neo4j database has a number of scalar datetime types that we use in the CE. Because of this, dates are provided as an object instead of a string value.

4.3.2 Updating a node

Query:

```
mutation {
 UpdateMusicComposition(
    identifier: "f7f4175a-714d-4b6f-81c1-c2bb8e52c1e4"
    name: "Symphony no1 in D major \"Titan\""
  ) {
    identifier
    name
    title
    creator
    created {
      formatted
    }
    modified {
      formatted
    }
    contributor
  }
}
```

Response:

```
{
  "data": {
    "UpdateMusicComposition": {
      "identifier": "f7f4175a-714d-4b6f-81c1-c2bb8e52c1e4",
      "name": "Symphony no1 in D major "Titan"",
      "title": "Symphony no. 1 in D major "Titan" - MusicBrainz",
      "creator": "https://github.com/trompamusic/trompa-ce-client/tree/master/demo",
      "created": {
        "formatted": "2021-03-30T09:37:50.363Z"
      },
      "modified": {
        "formatted": "2021-03-30T10:00:28.848Z"
      },
      "contributor": "https://musicbrainz.org"
    }
  }
}
```

Figure 4.7 Node update example

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.

4.3.3 Deleting a node

Query:

```
mutation {
 DeleteMusicComposition(
    identifier: "f7f4175a-714d-4b6f-81c1-c2bb8e52c1e4"
  ) {
    identifier
    name
  }
}
```

Response:

```
{
  "data": {
    "DeleteMusicComposition": {
      "identifier": "f7f4175a-714d-4b6f-81c1-c2bb8e52c1e4",
      "name": "Symphony no1 in D major "Titan""
    }
  }
}
```

Figure 4.8 Node deletion example

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 would be no node to delete:

Query:

```
mutation {
 DeleteMusicComposition(
    identifier: "f7f4175a-714d-4b6f-81c1-c2bb8e52c1e4"
  ) {
    identifier
    name
  }
}
```

Response:

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

Figure 4.9 Empty result examples

4.3.4 Add a relation between nodes

Query:

```
mutation {
 AddControlActionResult(
    from: { identifier: "0dc2c83b-c9d3-4b22-af6c-c4efe79f936f" }
    to: { identifier: "ff59650b-1d47-4ea5-b356-31fddeb48315" }
  ) {
    from {
      identifier
      __typename
      ... on ControlAction {
        result {
          identifier
          name
        }
      }
    }
    to {
      identifier
      __typename
    }
  }
}
```

Response:

```
{
  "data": {
    "AddControlActionResult": {
      "from": {
        "identifier": "0dc2c83b-c9d3-4b22-af6c-c4efe79f936f",
        "__typename": "ControlAction",
        "result": {
          "identifier": "ff59650b-1d47-4ea5-b356-31fddeb48315",
          "name": "'Tis by thy strength the mountains stand"
        }
      },
      "to": {
        "identifier": "ff59650b-1d47-4ea5-b356-31fddeb48315",
        "__typename": "DigitalDocument"
      }
    }
  }
}
```

Figure 4.10 Add relation example

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. Mutations that start with the word Add will create the same relation many times if called repeatedly, but mutations that start with Merge will only create the relation once.

A mutation to create 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.

4.3.5 Remove a relation between nodes

Query:

```
mutation {
 RemoveControlActionResult(
    from: { identifier: "0dc2c83b-c9d3-4b22-af6c-c4efe79f936f" }
    to: { identifier: "ff59650b-1d47-4ea5-b356-31fddeb48315" }
  ) {
    from {
      identifier
      __typename
      ... on ControlAction {
        result {
          identifier
          name
        }
      }
    }
    to {
      identifier
      __typename
    }
  }
}
```

Response:

```
{
  "data": {
    "RemoveControlActionResult": {
      "from": {
        "identifier": "0dc2c83b-c9d3-4b22-af6c-c4efe79f936f",
        "__typename": "ControlAction",
        "result": null
      },
      "to": {
        "identifier": "ff59650b-1d47-4ea5-b356-31fddeb48315",
        "__typename": "DigitalDocument"
      }
    }
  }
}
```

Figure 4.12 Remove relation example

The syntax for a mutation to remove a relation is identical to the Create mutation with a different name. A Remove mutation removes all the relations of the same type between the indicated nodes .

4.4. Subscriptions

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 are available. 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).

Query:

```
subscription {
 ThingCreateMutation(onTypes: [DigitalDocument]) {
    identifier
  }
}
```

Response:

```
Listening...
```

Figure 4.13 Subscription Query example

4.5 Authentication

All read operations in the CE-API are publicly accessible. However, in order to create, update, or delete nodes, the request **MUST BE** authenticated with a JWT token. All partners of the project will be provided with tokens with write access to the CE-API. When requested via support@videodock.com, third parties can be provided with tokens with write access.

The process of authenticating with the CE is described in greater detail in the GitHub documentation of the CE-API[20].

[20] https://github.com/trompamusic/ce-api/blob/0c3972da/docs/authentication.md

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]

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
}
```


Figure 5.1 REST interface example

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

6. Integration of software agents with the CE

The provisioned functionalities of the CE, as described and demonstrated in the previous chapters, allow for another type of 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.

The TROMPA Processing Library, as described in the D5.3 Trompa Processing Library[21] (TPL) deliverable, provides functionalities for descriptions and syntheses of music data coming from supported music data repositories. This comprises common access to (combinations of) public music data contained in the Contributor Environment (CE) database regardless of content-type and common access to WP3 algorithm processes to be run against (combinations of) this public music data. The TPL offers a communication layer between client applications, the CE, and WP3 technologies. Each time a client application requests any type of data or metadata, this is done via TPL functionalities which derive from:

- **The CE infrastructure** including the **CE neo4j database** along with the **GraphQL Interface,** which provides developers rich access to the TROMPA dataset, stored as public data references in the CE database, as well as the **Extended API Functionalities,** which allows for the real-time definition, creation, completion and consumption of WP3 technology 'jobs' on public music data referenced in the CE database.
- **The Multimodal Component,** which allows an application end-user to browse (and combine) CE references to public music data, regardless of content type. Moreover the Multimodal Component provides a graphical interface to browse the CE data.
- **A set of algorithms** that run on specific data types.
- Various instances of the **TROMPA Processing Component** that are run on different compute nodes and invoke different algorithms.
- **A client application** that makes algorithm invoking requests, e.g. ask to run a specific algorithm A on an item B. In the TROMPA context this client application is any of the five software prototypes developed to cater for TROMPA's use cases (see deliverables of WP6).

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

[21] https://trompamusic.eu/bibcite/reference/9

A subscription[22] mechanism, as described in section 4.4, can enable both the Component and Process systems 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-TPL interactions. This offers a standardized solution for integration and ensures TPL produced data is referenced and gets interlinked with the larger TROMPA dataset.

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

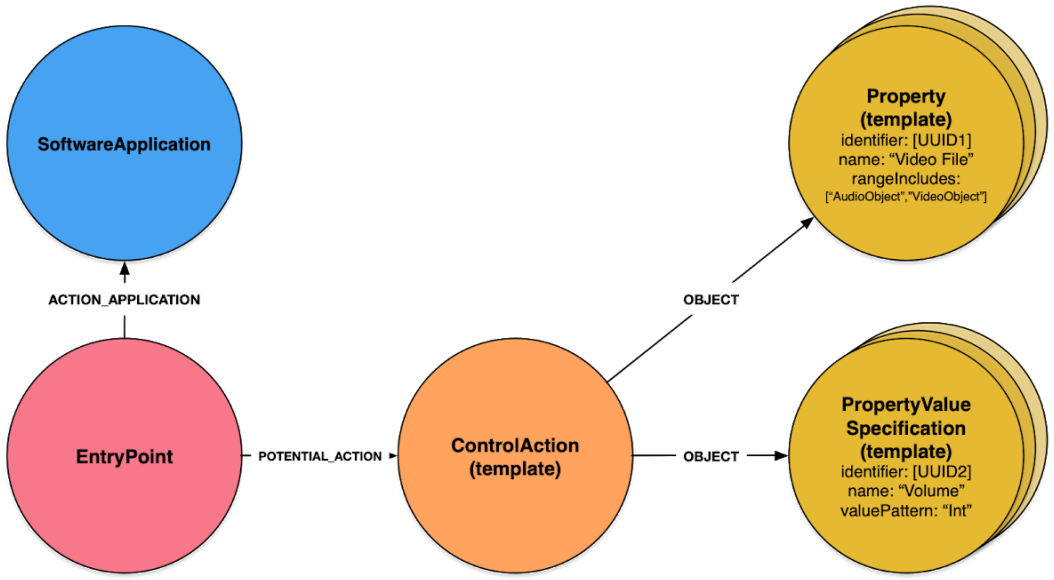

Figure 6.2 template nodes

Each WP3/4 process application, e.g. an alignment tool, will need a **SoftwareApplication**[23] 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**[24] node. This entry point corresponds to a user interface that enables a user to request, monitor and control the

[22] https://graphql.org/blog/subscriptions-in-graphql-and-relay/
[23] https://schema.org/SoftwareApplication
[24] https://schema.org/EntryPoint

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*[25] property.

The **ControlAction**[26] 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**[27] nodes and relating them to the **ControlAction** through the *object*[28] 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**[29] 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

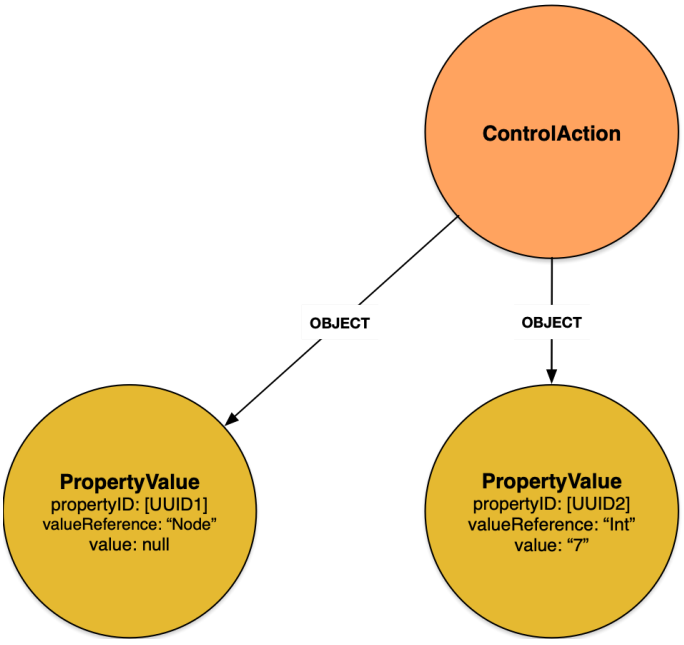

Figure 6.3 instance node example

[25] https://schema.org/actionApplication
[26] https://schema.org/ControlAction
[27] https://schema.org/PropertyValueSpecification
[28] https://schema.org/object
[29] https://meta.schema.org/Property

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.

Property nodes are always required in order to request a **ControlAction**. However, **PropertyValueSpecification** nodes can be optional when the *valueRequired* property is set to false. If the **PropertyValueSpecification** contains a *defaultValue*, this value will be used unless specified in the request. If the *valueRequired* is set to true and no *defaultValue* is specified, the response will yield an error when there is no value specified in the request.

If the RequestControlAction request passes validation, it will create a **ControlAction** node that is a copy of the 'template', plus one or more **PropertyValue**[30] 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.

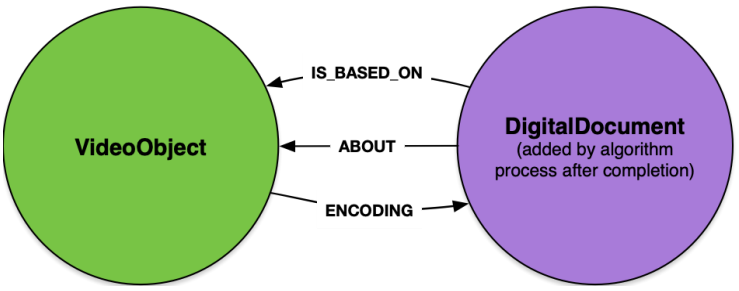

Figure 6.4 Public node

[30] https://schema.org/PropertyValue

6.2.4 End result

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

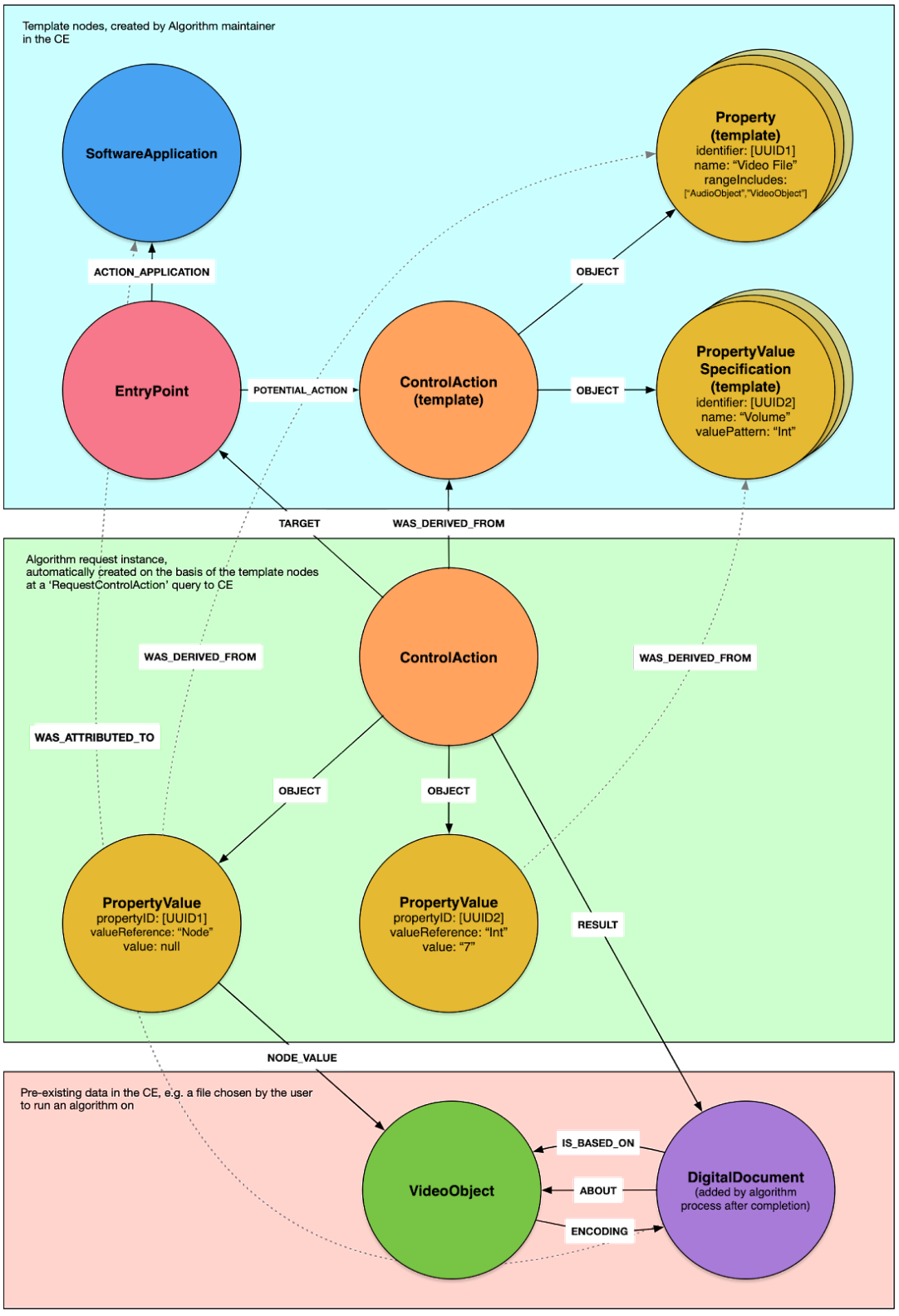

Figure 6.5 final collection of nodes

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 **EntryPoint**s (potential 'jobs'). An **EntryPoint** and related **Property** and **PropertyValueSpecification** nodes should contain enough information to dynamically set up a UI for a job request. The TPL can facilitate the creation of these nodes[31].

With the 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 **ControlAction**s with a custom 'ControlActionRequest' GraphQL subscription, and be immediately notified of a new request over a websocket connection.

After a new request comes 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[32] environment, which provides rich features for experimenting with and test GraphQL queries. Apollo[33] 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

[31] https://trompamusic.eu/deliverables/TR-D5.3-TROMPA_Processing_Library_v2.pdf
[32] https://github.com/prisma/graphql-playground#usage
[33] https://www.apollographql.com/docs/react/

Create **SoftwareApplication**

Query:

```
mutation {
 CreateSoftwareApplication(
    identifier: "ffb473fe-b345-4f10-8fee-424ef13f6686"
    contributor: "https://www.verovio.org"
    title: "Verovio MusicXML Converter"
    name: "Verovio MusicXML Converter"
    creator: "Verovio"
     description: "Verovio supports conversion from MusicXML to MEI. When converting from this web interface, the resulting MEI data will be displayed directly in the MEI-Viewer. The MEI file can be saved through the MEI  button that will be displayed on the top right."
    source: "https://github.com/rism-ch/verovio"
     subject: "Music notation engraving library for MEI with MusicXML,Humdrum support, toolkits, JavaScript, Python"
    format: "html"
    language: en
  ) {
    identifier
  }
}
```

Response:

```
{
  "data": {
    "CreateSoftwareApplication": {
      "identifier": "ffb473fe-b345-4f10-8fee-424ef13f6686"
    }
  }
}
```

Figure 6.6 Create **SoftwareApplication** example

This node will allow to group a number of **EntryPoint**s 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**

Query:

```
mutation {
 CreateEntryPoint(
    identifier: "d7a3b614-4c40-413f-99d6-c0da2c844963"
    contributor: "https://www.verovio.org"
    title: "Verovio MusicXML Converter"
    name: "Verovio MusicXML Converter"
    creator: "Verovio"
     description: "Verovio supports conversion from MusicXML to MEI. When converting from this web interface, the resulting MEI data will be displayed directly in the MEI-Viewer. The MEI file can be saved through the MEI  button that will be displayed on the top right."
    source: "https://github.com/rism-ch/verovio"
   subject: "Music notation engraving library for MEI with MusicXML,Humdrum support, toolkits, JavaScript, Python"
    format: "html"
    language: en
    actionPlatform: "TROMPA Algorithm Proof-Of-Concept"
    contentType: ["json"]
    encodingType: ["text"]
  ) {
    identifier
  }
}
```

Response:

```
{
  "data": {
    "CreateEntryPoint": {
      "identifier": "d7a3b614-4c40-413f-99d6-c0da2c844963"
    }
  }
}
```

Figure 6.7 Create **EntryPoint** example

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

Query:

```
mutation {
 AddEntryPointActionApplication (
    from: { identifier: "d7a3b614-4c40-413f-99d6-c0da2c844963" }
    to: { identifier: "ffb473fe-b345-4f10-8fee-424ef13f6686" }
  ) {
    from {
      identifier
      name
    }
    to {
      identifier
      name
    }
  }
}
```

Response:

```
{
  "data": {
    "AddEntryPointActionApplication": {
      "from": {
        "identifier": "d7a3b614-4c40-413f-99d6-c0da2c844963",
        "name": "Verovio MusicXML Converter"
      },
      "to": {
        "identifier": "ffb473fe-b345-4f10-8fee-424ef13f6686",
        "name": "Verovio MusicXML Converter"
      }
    }
  }
}
```

Figure 6.8

Create (template) **ControlAction**

Query:

```
mutation {
 CreateControlAction (
    identifier: "78d613b0-1064-4e9c-8f56-9c424d12bad9"
    description: "MusicXML to MEI conversion"
    name: "MusicXML to MEI conversion"
    actionStatus: PotentialActionStatus
  ) {
    identifier
    description
    actionStatus
  }
}
```

Response:

```
{
  "data": {
    "CreateControlAction": {
      "identifier": "78d613b0-1064-4e9c-8f56-9c424d12bad9",
      "description": "MusicXML to MEI conversion",
      "actionStatus": "PotentialActionStatus"
    }
  }
}
```

Figure 6.9

This **ControlAction** node will be the model for the 'job' created when a user does an algorithm process request. 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**

Query:

```
mutation {
 AddEntryPointPotentialAction(
    from: { identifier: "d7a3b614-4c40-413f-99d6-c0da2c844963" }
    to: { identifier: "78d613b0-1064-4e9c-8f56-9c424d12bad9" }
  ) {
    from {
      identifier
    }
    to {
      identifier
    }
  }
}
```

Response:

```
{
  "data": {
    "AddEntryPointPotentialAction": {
      "from": {
        "identifier": "d7a3b614-4c40-413f-99d6-c0da2c844963"
      },
      "to": {
        "identifier": "78d613b0-1064-4e9c-8f56-9c424d12bad9"
      }
    }
  }
}
```

Figure 6.10

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

Query:

```
mutation {
 CreateProperty(
    identifier: "2c796031-a303-460a-849d-0be95fb96b03"
    title: "MusicXML File"
    name: "targetFile"
    description: "Select a MusicXML file to be converted to MEI format"
    rangeIncludes: [DigitalDocument]
  ) {
    identifier
  }
}
```

Response:

```
{
  "data": {
    "MergeProperty": {
      "identifier": "2c796031-a303-460a-849d-0be95fb96b03"
    }
  }
}
```

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)

Query:

```
mutation {
 CreatePropertyValueSpecification (
    identifier: "f145799e-9612-43cb-9164-ac2d9ea2f460"
    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
  }
}
```

Response:

```
{
  "data": {
    "CreatePropertyValueSpecification": {
      "identifier": "f145799e-9612-43cb-9164-ac2d9ea2f460"
    }
  }
}
```

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**)

Query:

```
mutation {
 AddControlActionObject(
    from: { identifier: "78d613b0-1064-4e9c-8f56-9c424d12bad9" }
    to: { identifier: "f145799e-9612-43cb-9164-ac2d9ea2f460" }
  ) {
    from {
      __typename
    }
    to {
      __typename
      ... on PropertyValueSpecification {
        identifier
        title
      }
    }
  }
}
```

Response:

```
{
  "data": {
    "AddControlActionObject": {
      "from": {
        "__typename": "ControlAction"
      },
      "to": {
        "__typename": "PropertyValueSpecification",
        "identifier": "f145799e-9612-43cb-9164-ac2d9ea2f460",
        "title": "Result name"
      }
    }
  }
}
```

Figure 6.13

Query the resulting template model

Query:

```
query {
 EntryPoint(identifier: "d7a3b614-4c40-413f-99d6-c0da2c844963") {
    identifier
    title
    description
    contentType
    subject
    potentialAction {
      ... on ControlAction {
        identifier
        name
        actionStatus
        object {
          __typename
          ... on Property {
            identifier
            title
            description
            rangeIncludes
          }
          ... on PropertyValueSpecification {
            identifier
            title
            valueName
            valueRequired
            description
            defaultValue
            valueMinLength
            valueMaxLength
          }
        }
      }
    }
    actionApplication {
      identifier
      name
    }
  }
}
```

Response:

```
{
  "data": {
    "EntryPoint": [
      {
        "identifier": "d7a3b614-4c40-413f-99d6-c0da2c844963",
        "title": "Verovio MusicXML Converter",
        "description": "Verovio supports conversion from MusicXML to MEI. When converting
from this web interface, the resulting MEI data will be displayed directly in the
MEI-Viewer. The MEI file can be saved through the MEI  button that will be displayed on
the top right.",
        "contentType": [
          "json"
        ],
        "subject": "Music notation engraving library for MEI with MusicXML,Humdrum
support, toolkits, JavaScript, Python",
        "potentialAction": [
          {
            "identifier": "78d613b0-1064-4e9c-8f56-9c424d12bad9",
            "name": "MusicXML to MEI conversion",
            "actionStatus": "PotentialActionStatus",
            "object": [
              {
                "__typename": "PropertyValueSpecification",
                "identifier": "f145799e-9612-43cb-9164-ac2d9ea2f460",
                "title": "Result name",
                "valueName": "resultName",
                "valueRequired": true,
                "description": "What name would you like to give the result?",
                "defaultValue": "",
                "valueMinLength": 4,
                "valueMaxLength": 100
              },
              {
                "__typename": "Property",
                "identifier": "2c796031-a303-460a-849d-0be95fb96b03",
                "title": "MusicXML File",
                "description": "Select a MusicXML file to be converted to MEI format",
                "rangeIncludes": [
                  "DigitalDocument"
                ]
              }
            ]
          }
        ],
        "actionApplication": {
          "identifier": "ffb473fe-b345-4f10-8fee-424ef13f6686",
          "name": "Verovio MusicXML Converter"
        }
      }
    ]
  }
}
```


Figure 6.14

Leaving out the identifier from the query will list all available **EntryPoint**s, 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

Query:

```
subscription {
 ControlActionRequest(entryPointIdentifier: "d7a3b614-4c40-413f-99d6-c0da2c844963") {
    identifier
    actionStatus
    result {
      __typename
    }
    object {
      __typename
    }
  }
}
```

Response:

```
Listening...
```

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 **ControlAction**s created on the basis of a certain **EntryPoint** (*target* property) by adding a filter with target identifier:

Query:

```
query {
 ControlAction(
    filter: { target: { identifier: "558227e1-4a80-4097-899e-b13c5d581313" } }
  ) {
    identifier
    actionStatus
    target {
      __typename
      identifier
    }
  }
}
```

Response:


```
{
  "data": {
    "ControlAction": [
      {
        "identifier": "0dc2c83b-c9d3-4b22-af6c-c4efe79f936f",
        "actionStatus": "CompletedActionStatus",
        "target": {
          "__typename": "EntryPoint",
          "identifier": "558227e1-4a80-4097-899e-b13c5d581313"
        }
      },
      {
        "identifier": "21074faa-2752-40e4-9e62-ed748a2f9395",
        "actionStatus": "PotentialActionStatus",
        "target": {
          "__typename": "EntryPoint",
          "identifier": "558227e1-4a80-4097-899e-b13c5d581313"
        }
      }
    ]
  }
}
```


Figure 6.16

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

Query:

```
query {
 ControlAction(
    identifier: "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55"
  ) {
    identifier
    actionStatus
    object {
      __typename
      identifier
      ... on PropertyValue {
        name
        valueReference
        value
        nodeValue {
          __typename
          ... on DigitalDocument {
            identifier
            name
          }
        }
      }
    }
    result {
      __typename
    }
  }
}
```

Response:

```
{
  "data": {
    "ControlAction": [
      {
        "identifier": "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55",
        "actionStatus": "PotentialActionStatus",
        "object": [
          {
            "__typename": "PropertyValue",
            "identifier": "feb9e93c-672c-4485-a0de-986831486ed2",
            "name": "targetFile",
            "valueReference": "DigitalDocument",
            "value": null,
            "nodeValue": {
              "__typename": "DigitalDocument",
              "identifier": "4679dc75-11e4-41c7-b552-cd710df83dba",
              "name": "Dedham"
            }
          },
          {
            "__typename": "PropertyValue",
            "identifier": "99f22e3d-5c0f-4633-b21a-ed1991a13168",
            "name": "resultName",
            "valueReference": "String",
            "value": "Dedham MEI",
            "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**:

Query:

```
mutation {
 UpdateControlAction(
    identifier: "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55"
    actionStatus: ActiveActionStatus
  ) {
    identifier
    actionStatus
  }
}
```

Response:

```
{
  "data": {
    "UpdateControlAction": {
      "identifier": "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55",
      "actionStatus": "ActiveActionStatus"
    }
  }
}
```

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:

Query:

```
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: "MEI"
    contributor: "https://www.verovio.org"
    creator: "MusicXML to MEI converter",
    format: "mei",
    language: en,
    source: "https://example.domain/result-file.mei"
  ) {
    identifier
  }
}
```

Response:

```
{
  "data": {
    "CreateDigitalDocument": {
      "identifier": "57452f97-2d87-4018-85ce-56419688f8f1"
    }
  }
}
```


Figure 6.19

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

Query:

```
mutation {
 AddControlActionResult(
    from: {identifier: "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55" }
    to: { identifier: "57452f97-2d87-4018-85ce-56419688f8f1" }
  ) {
    from {
      __typename
    }
    to {
      __typename
    }
  }
}
```

Response:


```
{
  "data": {
    "AddControlActionResult": {
      "from": {
        "__typename": "ControlAction"
      },
      "to": {
        "__typename": "DigitalDocument"
      }
    }
  }
}
```


Figure 6.20

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

Query:

```
mutation {
 UpdateControlAction(
    identifier: "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55"
    actionStatus: CompletedActionStatus
  ) {
    identifier
    actionStatus
    result {
      __typename
      ... on DigitalDocument {
        identifier
        name
      }
    }
  }
}
```

Response:

```
{
  "data": {
    "UpdateControlAction": {
      "identifier": "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55",
      "actionStatus": "CompletedActionStatus",
      "result": {
        "__typename": "DigitalDocument",
        "identifier": "57452f97-2d87-4018-85ce-56419688f8f1",
        "name": "User defined name for the document"
      }
    }
  }
}
```

Figure 6.21

6.4 Third-party applications

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 applications could set up a subscription to the instantiated **ControlAction** via websocket to any mutations to the created job. The CE would notify the applications 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 applications 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[34] environment, which provides rich features for experimenting with and test GraphQL queries. Apollo[35] 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 **EntryPoint**s:

Query:

[34] https://github.com/prisma/graphql-playground#usage
[35] https://www.apollographql.com/docs/react/

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

Response:

```
{
  "data": {
    "EntryPoint": [
      {
        "identifier": "8ea5ecc3-d854-48ea-b0b7-b58d8309a615",
        "title": "Create User",
        "description": "This program creates a user model (inside the models folder) for
personalization",
        "contentType": [
          "application/json"
        ],
        "subject": "Emotion Recognition Model",
        "potentialAction": [
          {
            "identifier": "b20f00c7-73f1-4134-b0db-f8d170129a79",
            "name": "TPL Create User",
            "actionStatus": "PotentialActionStatus",
            "object": [
              {
                "__typename": "PropertyValueSpecification",
                "identifier": "0f0826f6-776b-45d0-a39e-63e4bd24347b",
                "title": "input_user",
                "valueName": "--input_user",
                "valueRequired": true,
                "description": "ID of the user",
                "defaultValue": "None",
                "valueMinLength": 0,
                "valueMaxLength": 100
              }
            ]
          }
        ],
        "actionApplication": {
          "identifier": "ee484daa-8951-4816-8e73-3fc831611ba8",
          "name": "Create User"
        }
      }
    ]
  }
}
```


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:

Query:

```
mutation {
 RequestControlAction(
    controlAction: {
      entryPointIdentifier: "d7a3b614-4c40-413f-99d6-c0da2c844963"
      potentialActionIdentifier: "78d613b0-1064-4e9c-8f56-9c424d12bad9"
      propertyObject: [
        {
          potentialActionPropertyIdentifier: "2c796031-a303-460a-849d-0be95fb96b03"
          nodeIdentifier: "4679dc75-11e4-41c7-b552-cd710df83dba"
          nodeType: DigitalDocument
        }
      ]
      propertyValueObject: [
        {
          potentialActionPropertyValueSpecificationIdentifier:
"f145799e-9612-43cb-9164-ac2d9ea2f460"
          value: "Dedham MEI"
          valuePattern: String
        }
      ]
    }
  ) {
    identifier
    __typename
  }
}
```

Response:

```
{
  "data": {
    "RequestControlAction": {
      "identifier": "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55",
      "__typename": "ControlAction"
    }
  }
}
```

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':

Query:

```
subscription {
 ControlActionMutation(identifier: "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55") {
    identifier
    actionStatus
    result {
      __typename
    }
    object {
      __typename
    }
  }
}
```

Response:

```
Listening...
```

Figure 6.24

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

Query:

```
subscription {
 ControlActionMutation(identifier: "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55") {
    identifier
    actionStatus
    result {
      __typename
    }
    object {
      __typename
    }
  }
}
```

Response:

```
{
  "data": {
    "ControlActionMutation": {
      "identifier": "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55",
      "actionStatus": "CompletedActionStatus",
      "result": {
        "__typename": "DigitalDocument"
      },
      "object": null
    }
  }
}
```


Figure 6.25

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

Query:

```
query {
 ControlAction(identifier: "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55") {
    identifier
    actionStatus
    result {
      __typename
    }
    object {
      __typename
    }
  }
}
```

Response:

```
{
  "data": {
    "ControlAction": [
      {
        "identifier": "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55",
        "actionStatus": "CompletedActionStatus",
        "result": {
          "__typename": "DigitalDocument"
        },
        "object": [
          {
            "__typename": "PropertyValue"
          },
          {
            "__typename": "PropertyValue"
          }
        ]
      }
    ]
  }
}
```


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':

Query:

```
mutation {
 UpdateControlAction(
    identifier: "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55"
    actionStatus: CompletedActionStatus
  ) {
    identifier
    actionStatus
  }
}
```

Response:

```
{
  "data": {
    "UpdateControlAction": {
      "identifier": "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55",
      "actionStatus": "CompletedActionStatus"
    }
  }
}
```


Figure 6.27

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

Query:

```
query {
 ControlAction(identifier: "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55") {
    identifier
    actionStatus
    result {
      __typename
      ... on DigitalDocument {
        identifier
        name
        source
      }
    }
    object {
      __typename
    }
  }
}
```

Response:

```
{
  "data": {
    "ControlAction": [
      {
        "identifier": "9e4d6858-f7c5-43bb-82fc-ed23aea4bb55",
        "actionStatus": "CompletedActionStatus",
        "result": {
          "__typename": "DigitalDocument",
          "identifier": "57452f97-2d87-4018-85ce-56419688f8f1",
          "name": "User defined name for the document",
          "source": "https://example.domain/result-file.mei"
        },
        "object": [
          {
            "__typename": "PropertyValue"
          },
          {
            "__typename": "PropertyValue"
          }
        ]
      }
    ]
  }
}
```


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 **ControlAction**s 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 to 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. 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).

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 and other contributors to develop their integration with the TROMPA CE and develop prototypes and other software based on TROMPA infrastructure.

8. References

8.1 List of abbreviations

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	Voctro 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

A. Appendix A: List of core metadata properties and examples

Property	Type	On	Example(s)	Explanation
title	String	thing	'Symfony No. 2'	The contents of the HTML <title> tag of the web resource which describes the entity. 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.
creator	String	thing	'https://github.com/trompamusic/ce-data-import'	A URL representing the agent who created the node, either a link to the source code of a software tool, or the WebID of a Solid user.
			'Gustav Mahler'	The creator name should not be a natural name
			'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
subject	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	Stick to keywords only. The *description* property allows a flowing test.

			Linz...'	
description	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 *subject* property
publisher	String	thing	'Friedrich Hofmeister, Leipzig'	The person, organization or service responsible for making available the thing the web resource is about. Typically, the name of the Publisher should be used, if possible with a place indication. 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.
			'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.
contributor	URL	resource	'https://imslp.org'	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. 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. 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 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
date	Date	thing	‘1895-12-13’	A point in time associated with an event in the lifecycle of the resource. Must be in ‘YYYY-MM-DD’ format. Examples: composition first performance, publishing date, birth date, release date, file generation date, annotation date
			‘1895’	Is not a valid date
			‘2018-12-04 12:04:11 ’	Is not a valid Date
type	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 *additionalType* property.
			‘mo:composition’	Turtle prefix notation is not supported
			‘’	Can be left empty
format	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 (https://www.iana.org/assignments/media-types/media-types.xhtml) If necessary, we can define a custom mime type (vnd.trompamusic.[type])

identifier	UUID	CE	‘5d05bfda-c050-424e-9d11-314b80225ea8’	An unambiguous reference to the resource within a given context. For CE we will use UUID. An non-unique UUID will fail validation. If no UUID is passed, one will be generated by the CE (recommended).
			‘’	When left empty, the CE will generate a UUID (recommended)
source	URL	resource	‘https://imslp.org/wiki/Symphony_No.2_(Mahler,_Gustav)’	The URL of the web resource to be represented by the node.
			‘https://api.trompamusic.eu/b670c593-d1c6-45ed-9a09-6a51c12108e1‘	Any TROMPA produced data stored in CE will automatically have a unique URl 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
language	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
inLanguage	enum	metadata	‘en’	The language of the content (for example lyrics).

			‘en’	If the work has no language content (e.g. an instrumental composition), the field should be null.
contentUrl	URL	resource	https://raw.githubusercontent.com/trompamusic-encodings/Beethoven_Op35_BreitkopfHaertel/master/Beethoven_Op35.mei	URL of the raw bytes of a media resource on the Web
			arcp://ni,sha-256;ZjBhOTYyZTUxYjUzYzlmNWRlOTJjMTNlMTkxZGM4N2NjMGYyYTQzNTc0NTkxMWI0ZDU4YTUyNDQ2Mjk5YmUxNg==/07_affer_opem.xml	ARCP-protocol URL used to address a media resource contained inside a compressed archive
source	URL	resource	https://github.com/trompamusic-encodings/Beethoven_Op35_BreitkopfHaertel/blob/master/Beethoven_Op35.mei	URL of a web resource describing (or providing an HTML view of) the contentUrl resource
			https://imslp.org/wiki/File:PMLP580712-07_affer_opem.zip	URL of a web resource describing (or providing an HTML view of) the contentUrl resource
url	URL	resource	https://github.com/trompamusic-encodings/Beethoven_WoO80_BreitkopfHaertel	Persistent URI representing the resource identified by the node
			https://imslp.org/wiki/Special:ReverseLookup/359599	Persistent URI representing the resource identified by the node