Consiglio Nazionale delle Ricerche
Istituto di Matematica Applicata
e Tecnologie Informatiche
“Enrico Magenes”

REPORT SERIES

A. Repetto, C. E. Catalano, M. Spagnuolo

Architecture of a client-server platform for IMATI cultural heritage applications

19-05

IMATI Report Series Nr. 19-05
24 July, 2019

Architecture of a client-server platform for IMATI cultural heritage applications

Andrea Repetto, Chiara Eva Catalano, Michela Spagnuolo

Andrea Repetto
Istituto di Matematica Applicata e Tecnologie Informatiche "E. Magenes"
Consiglio Nazionale delle Ricerche
Via De Marini, 6 - 16149 Genova, Italy
Email: andrea.repetto@ge.imati.cnr.it

Chiara Eva Catalano
Istituto di Matematica Applicata e Tecnologie Informatiche "E. Magenes"
Consiglio Nazionale delle Ricerche
Via De Marini, 6 - 16149 Genova, Italy
Email: chiara.catalano@ge.imati.cnr.it

Michela Spagnuolo
Istituto di Matematica Applicata e Tecnologie Informatiche "E. Magenes"
Consiglio Nazionale delle Ricerche
Via De Marini, 6 - 16149 Genova, Italy
Email: michela.spagnuolo@ge.imati.cnr.it

Abstract.

The final technological outcome of the GRAVITATE EU project is a software platform for the analysis and restoration of cultural heritage artefacts. The GRAVITATE platform is a client-server platform, composed of a back-end infrastructure, of a web interface and a desktop interface. The role of the web interface is to offer browsing and semantic and geometric similarity capabilities, while the desktop client is designed for graphical interaction and manipulation of high-resolution models. Both clients connect to a back-end infrastructure that offers access to 3D models and their geometric properties using a RESTful API.
After the end of the project, a work of redesign of the original platform has been carried out, to exploit IMATI applications developed in the project, make them independent of the GRAVITATE servers and services, and putting the basis for future research in the CH domain. This technical report describes the internal organization of the client-server platform that was developed by reusing the parts already developed, and reimplementing the back-end and web interface.
The back-end and the web interface were implemented in Python, using the Django framework, while the desktop interface was developed in C++, with the Qt library for the user interface. This technical report describes also the code organization and guides the developer in the task of performing changes and extensions to the platform.

Keywords: *Client-server platform, 3D semantic annotation, Geometric similarity, Cultural Heritage*

[page left intentionally blank]

Architecture of a client-server platform for IMATI cultural heritage applications

Andrea Repetto, Chiara Eva Catalano, Michela Spagnuolo

Abstract

The final technological outcome of the GRAVITATE EU project is a software platform for the analysis and restoration of cultural heritage artefacts. The GRAVITATE platform is a client-server platform, composed of a back-end infrastructure, of a web interface and a desktop interface. The role of the web interface is to offer browsing and semantic and geometric similarity capabilities, while the desktop client is designed for graphical interaction and manipulation of high-resolution models. Both clients connect to a back-end infrastructure that offers access to 3D models and their geometric properties using a RESTful API.

After the end of the project, a work of redesign of the original platform has been carried out, to exploit IMATI applications developed in the project, make them independent of the GRAVITATE servers and services, and putting the basis for future research in the CH domain. This technical report describes the internal organization of the client-server platform that was developed by reusing the parts already developed, and reimplementing the back-end and web interface.

The back-end and the web interface were implemented in Python, using the Django framework, while the desktop interface was developed in C++, with the Qt library for the user interface. This technical report describes also the code organization and guides the developer in the task of performing changes and extensions to the platform.

Keywords: Client-server platform, 3D semantic annotation, Geometric similarity, Cultural Heritage

Table of contents

1 Introduction

The work described in this report is mainly based on the GRAVITATE platform, the main software product developed during the GRAVITATE project. The GRAVITATE platform is a client-server software platform for the management and analysis of Cultural Heritage collections [1,2]. The platform is composed of a back-end component, which organizes the access to the 3D data, the cultural heritage metadata and a selection of geometric measurements which have been considered useful in this application scenario , and two front-end components, one as a desktop application, and one as a web application. Both clients connect to the same back-end.

The GRAVITATE platform was developed in collaboration with other partners of the project: mainly IT Innovation (back-end and web front-end) and IMATI (desktop front end, back-end). Some functionalities included in the platform have been developed by IMATI, UvA (faceting algorithm) and Technion/Haifa (ReAssembly algorithm).[1] In particular, the parts entirely developed by IMATI are the following:

- The Desktop Client [3];
- The Gravifix tool [4];
- The SimilarityLib algorithms [5];
- The Similarity Search engine [6–8].
- The Feature Extraction algorithm [9];

Both the back-end and the web-based client have been developed as extensions of the ResearchSpace platform. ResearchSpace is a platform developed by British Museum, built to provide access to the BM repository, with a focus on Semantic Web technologies (RDF, SPARQL)[2].

At the end of the project, it was considered useful to reintegrate the parts already developed by IMATI during the project, and provide an in-house reimplementation of some parts developed by other partners. The choice of developing a platform containing only IMATI software has the advantage of having full control on the development cycle, and facilitates future research works in the Cultural Heritage field.

This document guides the developer to the set up and maintenance of the IMATI client-server architecture.

The document is structured as follows: Section 1 is this section; Section 2 is an explanation of the web-based platform, composed of a back-end and a front-end, which been included in the same codebase for simplicity and easier maintainability. Section 2.7 describes the internal organization of the Desktop Client code. Section 4 contains some guidelines on the implementation of an annotation back-end, a functionality that did not make it in the current release of the platform.

[1] the faceting algorithm and the ReAssembly algorithm have not been integrated in the platform, and run as separate executables

[2] https://www.w3.org/TR/rdf11-primer/

2 Web-based platform (back-end and front-end)

The outcome of the GRAVITATE project, in terms of software, consists of a software platform with a client-server architecture; besides, the clients are two: a web client and a desktop client. Both clients are dependent on a back-end server. A synchronization mechanism between the two clients happens through a component called Clipboard. The Clipboard is a shared space where the users could save the fragments of interest, to be used for the working session. The two clients were developed by different partners: the desktop client by IMATI, and the web client by IT Innovation; also, IT Innovation took care of developing and deploying the back-end of the platform.

Both the web client and the back-end are heavily dependent on the ResearchSpace platform, whose main developer is British Museum. Despite its many built-in functionalities, it has proven difficult to maintain and customize to the needs of IMATI future research projects. Thus, it has been considered important to consolidate the already developed software in a full-featured platform, to support further developments and extensions. In order to do this, a drop-in replacement for the existing back-end has been designed and implemented.

2.1 Docker container organization

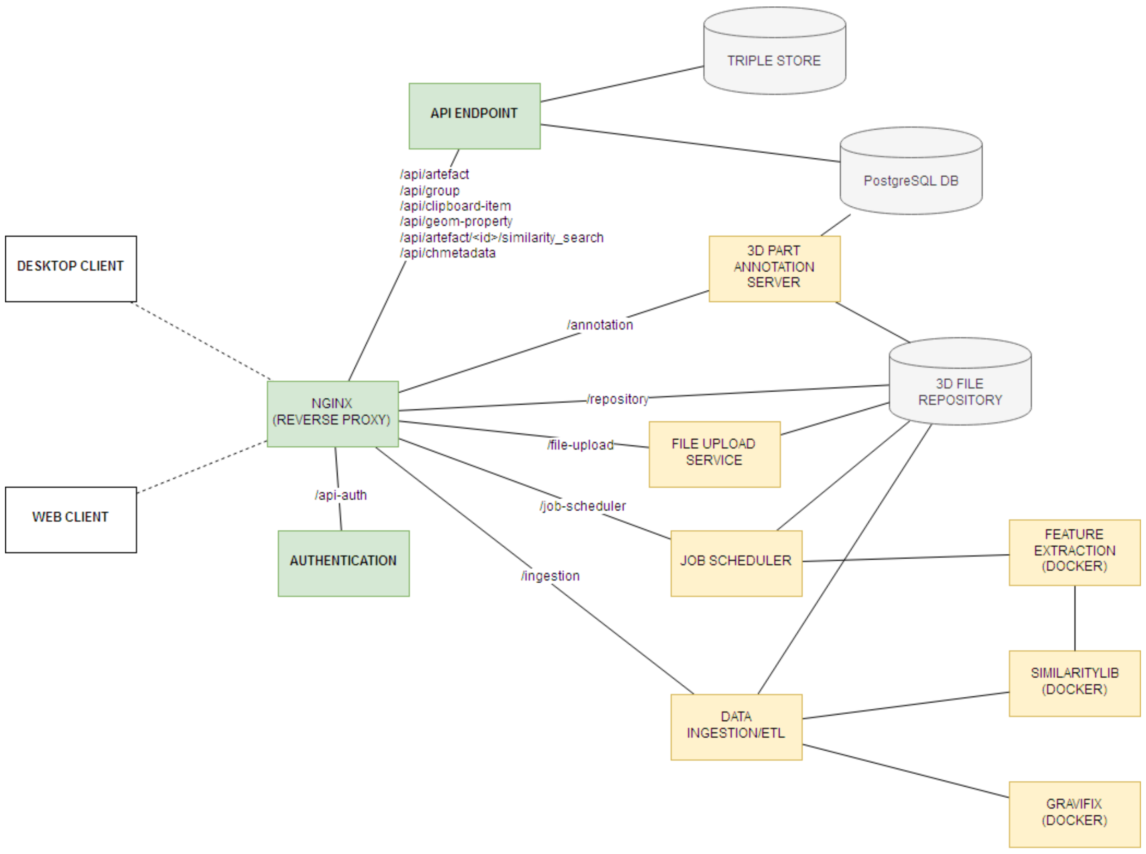


Figure 1: Architecture of the IMATI back-end; the services in green are those currently implemented, while the services in yellow do not have an implementation yet

The architecture is organized as a set of Docker[3] services (Figure 1). Docker is a virtualization tool that adds a layer of abstraction over the Linux operating system. Docker is built around the concept of containers: each container is an isolated operating system environment, which can be configured to run a specific application (such as, a web server, a database management system, a RESTful API service).

When a container is run, it is called a "service". Each Docker service has a distinct network interface, and it is possible to set up a private virtual network to make them communicate through the TCP/IP protocol. Services can also

[3] https://www.docker.com

communicate with the external network, such as, accepting requests from hosts on the Internet, or sending requests to external services.

The architecture chosen for the back-end is modular, therefore it is possible to add new services without much effort, or even to replace the existing ones with drop-in replacements, requiring only minimal changes to the rest of the architecture (for example, changing the DBMS from PostgreSQL to MySQL).

Here we describe the modules composing the architecture:

- **NGINX (REVERSE PROXY):** NGINX[4] is a web server with reverse proxy functionalities, therefore it is possible to redirect some requests matching a specific pattern to a host of the network, and return the responses to the client that requested them. For example, all the requests matching the **`/api/`** pattern will be redirected to the "API ENDPOINT" module. The configuration of the NGINX service is in the **`nginx.conf`** file in the root of the code repository.
- **API ENDPOINT**: This endpoint offers all the basic CRUD (Create, Read, Update, Delete) operations for the database, so that the user is able to search and retrieve information related to an artefact, to the user clipboard, and also to perform content-based retrieval. The API endpoint is implemented in Python, using Django Rest Framework[5], a library that extends Django[6], a mature and robust web development framework.
- **PostgreSQL[7] DB**: the database that stores all the data related to the artefacts, the users' clipboards, and also the similarity metrics that are used for the search engine. The schema of the database (except the search engine) is represented in Figure 3.
- **Triple store**: an additional graph database based on the RDF standard, which is mainly used to store semantic metadata related to the artefacts, and the 3D part annotations (expressed as a set of triples); a recommended triple store is Blazegraph[8], which was already used in the GRAVITATE project, being part of the ResearchSpace platform.
- **AUTHENTICATION**: Conceptually, this is a service providing some external authentication method. Currently, the API ENDPOINT service takes care of the authentication as well.
- **3D File Repository**: a file system organized according to the GRAVITATE repository file organization. The files in the file system are not deployed in any container; using the "volume" mechanism provided by Docker, a specific directory is mounted as the "repository" volume, so that other services can access these files through the **`/repository`** path.

Other modules have been included in the design in order to have a fully functional architecture, but do not have an implementation yet:

- **Job Scheduler**: a service that handles requests for algorithms which have long execution times (in the order of minutes). The scheduler takes care of monitoring the execution state of the algorithm, and to return the results of the algorithm when finished, or an error message when the execution fails.
- **3D Part annotation server**: a service that provides part annotation functionalities according to the extension of the Web Annotation Data Model for 3D models. This service is able to recognise and store 3D selectors related to meshes (**ThreeDMeshPointSelector**, **ThreeDMeshLineSelector**, **ThreeDMeshSurfaceSelector**) [10]. The specification of the service API is described in the Web Annotation Protocol[11] document, and a guideline for its implementation is provided.
- **GraviFix[9], SimilarityLib[10], FeatureExtraction[11]:** these are on-demand services, run by other services to perform specific operations or algorithms. The results can be either returned to the user (**FeatureExtraction**) or stored in the database after a batch processing on multiple files (**GraviFix**, **SimilarityLib**).

4 https://nginx.org
5 https://www.django-rest-framework.org/
6 https://www.djangoproject.com/
7 https://www.postgresql.org/
8 https://www.blazegraph.com/
9 https://github.com/CNR-IMATI/GraviFix
10 https://github.com/CNR-IMATI/SimilarityLib
11 https://github.com/CNR-IMATI/FeatureExtraction

- **Data Ingestion:** this service provides the capability of *ingesting* a new 3D artefact into the platform. Once the "original" 3D model is uploaded, it calls the **GraviFix** and **SimilarityLib** services to produce the derivative files, which are simplified versions of the original model, and also to populate the database with corresponding metadata. This services takes care of computing also the similarity distances for the similarity search engine.

The set-up of the services is achieved through the **docker-compose** tool. The main configuration file is called **`docker-compose.yml`**: it specifies which Docker containers should be instantiated, and how they communicate with each other.

The back-end can be executed by running the command:

```
docker-compose up -d
```

from the root directory of the project.

It is possible to inspect the logs with:

```
docker-compose logs -f
```

NOTE: The detailed instructions to set up and run the back-end (either with development and production configurations) are described in the README.md file in the root directory of the *imati-gravitate-backend*[12] repository.

2.2 Populating the database and the file repository

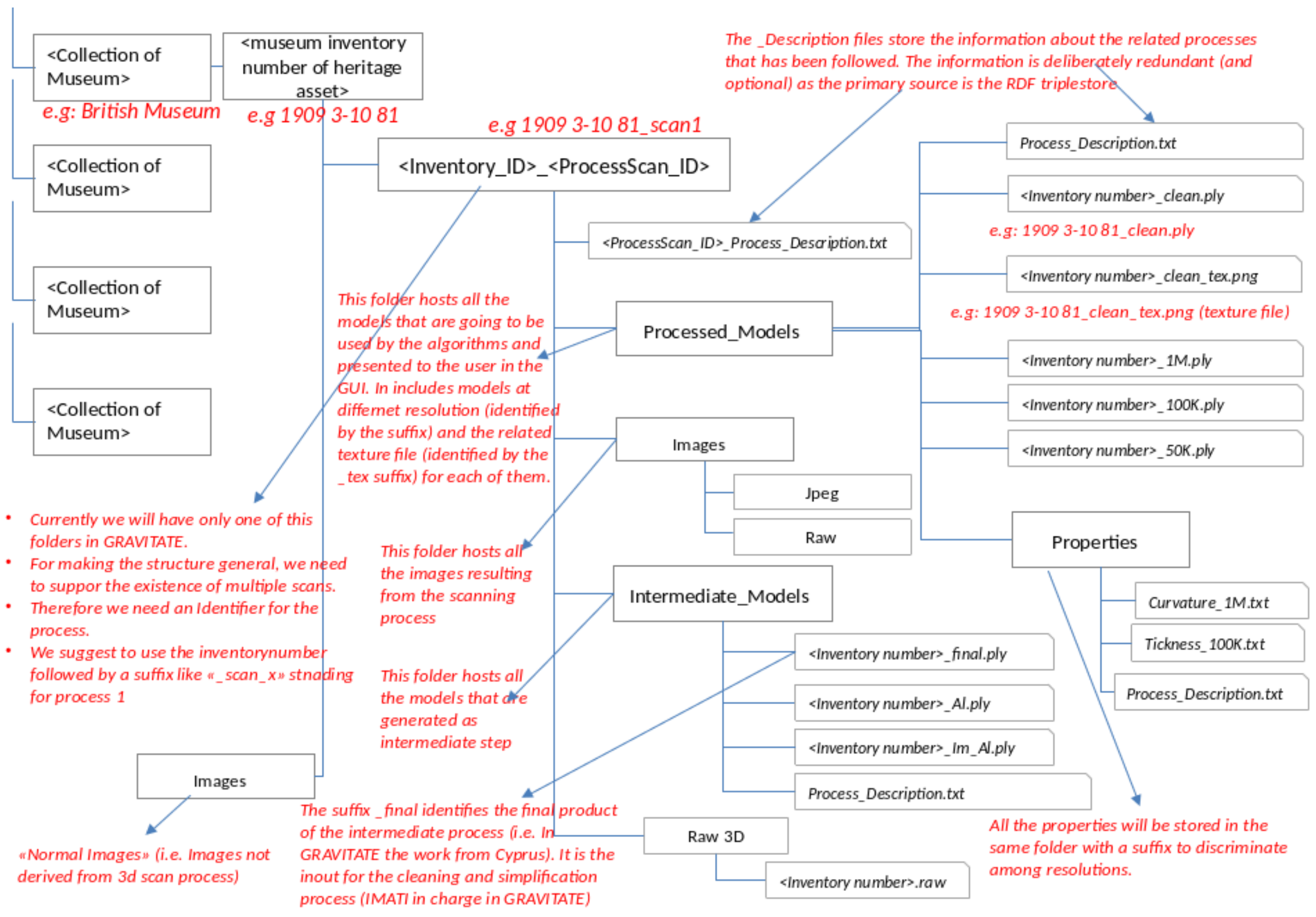
In order to make the platform operational in the shortest time, the fully automated ingestion pipeline has not been implemented, since it was not implemented in the GRAVITATE platform itself; instead, a simplified version has been made, which uses pre-computed data where possible. These pre-computed data include:

- Simplified 3D models: clean, 1M, 100K and 50K versions (computed with GraviFix);
- Geometric properties: Mean Curvature, Shape Index, SDF, LAB color, Area, Volume, Average Thickness, Facet files (computed with SimilarityLib);
- Similarity matrices, for the Salamis and Naukratis datasets (computed with SimilarityLib).

The 3D model repository is populated via a script called **import_repository_files**. It is an administration script implemented as part of the Django server: the advantage of this approach is that the database connection layer provided by Django can be reused as it is. The script takes as input a series of directories containing 3D models in PLY format and pre-computed geometric properties, plus a table in CSV format containing the data to populate the **artefacts** table.

The output is a file system organized according to the latest repository structure of the GRAVITATE platform [4,12], see Figure 2. The script populates also the **geometric_properties** table, ensuring that the relations between the two tables are set up properly.

[12] https://github.com/andrea-repetto/imati-gravitate-backend



<Collection of Museum>
e.g: British Museum
<museum inventory number of heritage asset>
e.g 1909 3-10 81
e.g 1909 3-10 81_scan1
<Inventory_ID>_<ProcessScan_ID>
<Collection of Museum>
<Collection of Museum>
<Collection of Museum>
The _Description files store the information about the related processes that has been followed. The information is deliberately redundant (and optional) as the primary source is the RDF triplestore
Process_Description.txt
<Inventory number>_clean.ply
e.g: 1909 3-10 81_clean.ply
<ProcessScan_ID>_Process_Description.txt
<Inventory number>_clean_tex.png
e.g: 1909 3-10 81_clean_tex.png (texture file)
This folder hosts all the models that are going to be used by the algorithms and presented to the user in the GUI. In includes models at differnet resolution (identified by the suffix) and the related texture file (identified by the _tex suffix) for each of them.
Processed_Models
<Inventory number>_1M.ply
<Inventory number>_100K.ply
<Inventory number>_50K.ply
Images
Jpeg
Raw
Properties
Currently we will have only one of this folders in GRAVITATE.
For making the structure general, we need to suppor the existence of multiple scans.
Therefore we need an Identifier for the process.
We suggest to use the inventorynumber followed by a suffix like «_scan_x» stnading for process 1
This folder hosts all the images resulting from the scanning process
Intermediate_Models
Curvature_1M.txt
Tickness_100K.txt
Process_Description.txt
<Inventory number>_final.ply
This folder hosts all the models that are generated as intermediate step
<Inventory number>_AI.ply
<Inventory number>_Im_AI.ply
Process_Description.txt
Images
The suffix _final identifies the final product of the intermediate process (i.e. In GRAVITATE the work from Cyprus). It is the inout for the cleaning and simplification process (IMATI in charge in GRAVITATE)
Raw 3D
<Inventory number>.raw
«Normal Images» (i.e. Images not derived from 3d scan process)
All the properties will be stored in the same folder with a suffix to discriminate among resolutions.

Figure 2: The latest version of the GRAVITATE repository structure; the output of the ***repo-organizer*** *script is compliant with this specification*

Another administration command that is provided is **import_similarity_matrices**: this populates the similarity matrices using the matrices provided in textual form, in the datasets/SALAMIS and datasets/NAUKRATIS directory. The datasets.json file provides the association between the matrix files and the corresponding descriptors (listed in Section 2.3.6). Matrix files are simple text files with space-separated values arranged as a matrix. The number of rows and columns depends on the dataset. The association between the inventory IDs and the positions in the matrix is in the file **`<DATASET>_INVENTORY_ID.txt`**. This file is a simple list of inventory IDs, where the i-th element in the list corresponds to the i-th row/column in the matrix.

The whole procedure, consisting of the database initialization, the population of the repository and the population of the similarity matrix can be called with a single command:

`./init_and_populate.sh`

This command will perform the whole procedure with the default datasets (currently, Salamis, Naukratis and the Gath Jars). In order for the administration commands to work properly, they are launched inside the appropriate docker container (**`django-apps`**).

This is the content of the **`init_and_populate`** script:

```
#!/bin/bash

MANAGE_CMD="docker-compose run --rm django-apps python manage.py"

# ensure no docker container is running
docker-compose down
```

```
# Initializes the database (runs the migration scripts)
$MANAGE_CMD migrate

# Imports artefacts and geometric properties in the repository
$MANAGE_CMD import_repository_files -i "/gravitate-import/SALAMIS_MODELS" \
            -i "/gravitate-import/NAUKRATIS_MODELS" \
            -i "/gravitate-import/HAIFA_TECHNION_MODELS" \
            -t "/gravitate-import/THUMBNAILS" \
            -m "/datasets/artefact-table.csv" \
            -o "/repository/"

# Imports the similarity matrices
# NOTE: it is required that the "artefacts" table has been populated first
$MANAGE_CMD import_similarity_matrices "/datasets/datasets.json"
```

Single administration commands (e.g. **`migrate`**, **`import_repository_files`**, **`import_similarity_matrices`**) can be run with the django-admin.sh, which is just a shortcut to the **django-apps/manage.py** script, run inside its Docker container[13].

2.3 Database schema

This section explains the schema of the relational database that stores all the information. Besides these tables, there are the default tables created by Django, which manage users, user groups with different privileges and sessions. Indeed, the **`users`** table depicted in Figure 3 is the **`django_users`** table.

NOTE: all the tables pertaining to the *backend* app have a "**`backend_`**" prefix. For instance, the **Artefact** model class in the "backend" app is mapped to the **`backend_artefact`** table. The prefix is added automatically by Django to avoid conflicting names in other apps.

[13] Note that the administration commands are always meant to run inside the docker container because the whole configuration of the django-apps project is based on the paths and database connections provided by the docker-compose configuration file

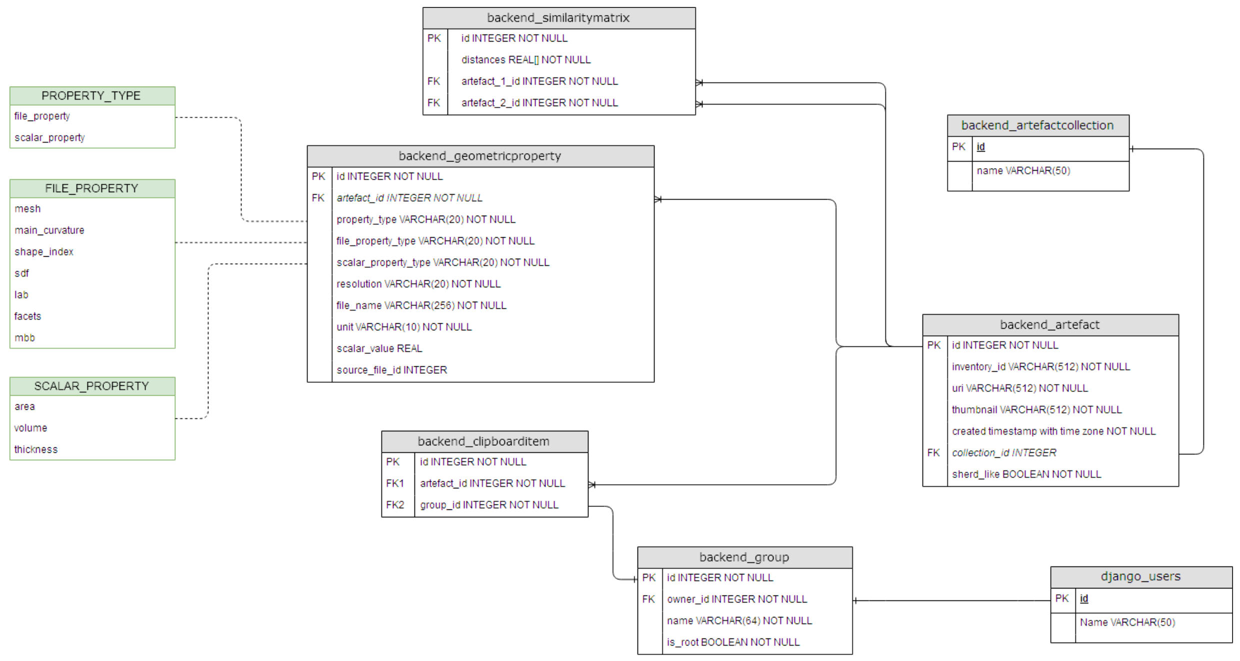

backend_similaritymatrix
PK id INTEGER NOT NULL
distances REAL[] NOT NULL
FK artefact_1_id INTEGER NOT NULL
FK artefact_2_id INTEGER NOT NULL
PROPERTY_TYPE
file_property
scalar_property
FILE_PROPERTY
mesh
main_curvature
shape_index
sdf
lab
facets
mbb
SCALAR_PROPERTY
area
volume
thickness
backend_geometricproperty
PK id INTEGER NOT NULL
FK artefact_id INTEGER NOT NULL
property_type VARCHAR(20) NOT NULL
file_property_type VARCHAR(20) NOT NULL
scalar_property_type VARCHAR(20) NOT NULL
resolution VARCHAR(20) NOT NULL
file_name VARCHAR(256) NOT NULL
unit VARCHAR(10) NOT NULL
scalar_value REAL
source_file_id INTEGER
backend_artefactcollection
PK id
name VARCHAR(50)
backend_artefact
PK id INTEGER NOT NULL
inventory_id VARCHAR(512) NOT NULL
uri VARCHAR(512) NOT NULL
thumbnail VARCHAR(512) NOT NULL
created timestamp with time zone NOT NULL
FK collection_id INTEGER
sherd_like BOOLEAN NOT NULL
backend_clipboarditem
PK id INTEGER NOT NULL
FK1 artefact_id INTEGER NOT NULL
FK2 group_id INTEGER NOT NULL
backend_group
PK id INTEGER NOT NULL
FK owner_id INTEGER NOT NULL
name VARCHAR(64) NOT NULL
is_root BOOLEAN NOT NULL
django_users
PK id
Name VARCHAR(50)

Figure 3: Diagram of the database schema. Note: the green boxes indicate the list of accepted values for the corresponding field

2.3.1 "Artefact" table

The **backend_artefact** table contains the metadata related to each artefact:

- **id** is the primary key of the table;
- **inventory_id** is a string that acts as an inventory number for the archaeological artefact (e.g. D292, GR_10_1890 and so on);
- **uri** is the semantic URI of the fragment (to maintain compatibility with the old platform);
- **thumbnail** is the relative path of the thumbnail in the file repository;
- **created** is a timestamp of the record creation (i.e. when the artefact has been inserted in the database);
- **dataset_name** is the name of the original dataset (e.g. SALAMIS, NAUKRATIS). It is used to provide customized parameters according to the dataset used, in the similarity search API;
- **sherd_like** is a Boolean attribute denoting artefacts with a "sherd-like" shape. It is used in the similarity search interface, since a warning is displayed when activating the "thickness" descriptor (the thickness values are not meaningful for non-sherd-like artefacts).

2.3.2 "Artefact collection" table

The **backend_artefactcollection** table organizes information pertaining to the artefact collections. They are used in the web interface as a filter for the "browsing" page, so that only the items belonging to a specific collection are returned.

The table contains the following items:

- **id** is the primary key of the table
- **name** is the name of the collection

2.3.3 "Geometric Property" table

The **backend_geometricproperty** table contains information on the geometric properties of artefacts.

Geometric properties can be of two categories: scalars and files. Scalars are **area**, **volume** and **thickness**, while files are the remaining ones: **meshes**, **scalar field properties**, **facet** files (fct), and the **bounding box file.**

The fields of this table are the following:

- **id** is the primary key of the table;
- **artefact_id**: the ID of the corresponding artefact;

- **property_type**: the property type (scalar or file);
- **file_property_type**: the subtype of a property of "file" type;
- **scalar_property_type**: the subtype of a property of "scalar" type;
- **resolution**: the resolution of the property (one of **`original`**, **`clean`**, **`1M`**, **`100k`**, **`50k`**);
- **file_name**: the file name if the property is of "file" type; blank otherwise;
- **unit**: the unit of measure, used both for file and scalar properties;
- **scalar_value**: the scalar value, for a property of "scalar" type; NULL otherwise;
- **source_file**: the original file from which the property has been computed[14].

NOTE: The fields **property_type**, **file_property_type** and **scalar_property_type** have type **`VARCHAR(20)`** (i.e. they are strings of variable length, maximum 20), but they should be regarded as ENUM types. Although PostgreSQL would support the ENUM type, the Django ORM does not support it, and uses the VARCHAR type instead[15].

2.3.4 "Group" table

The **`backend_group`** table contains the groups of artefacts created by the users; the only actual content of the group is the name, while the group content is stored in the clipboard-items table. Each group has an owner, because the clipboard API retrieves only the groups belonging to the logged in user.

The table has the following rows:

- **id** is the primary key of the table;
- **name** is the name of the group;
- **owner_id** is the user ID of the owner of the group;
- **is_root** is a Boolean value that states whether the group is the "root" of the hierarchy or not.

2.3.5 "Clipboard-item" table

The **`backend_clipboarditem`** table maintains the association between artefacts and groups, i.e. which items belong to each group.

- **id** is the primary key of the table;
- **artefact_id** is the ID of the artefact (foreign key to the artefacts) table;
- **group_id** is the ID of the group (foreign key to the groups table).

NOTE: there is no actual constraint on the uniqueness of an artefact inside a specific group, as there was not in the original GRAVITATE platform: the same artefact may occur multiple times.

2.3.6 "Similarity-matrix" table

NOTE: this section only describes the schema of the table; for an in-depth explanation of the similarity search functionality, please refer to Section 2.4.

The **`backend_similaritymatrix`** table contains measures of distance between the artefacts in the repository:

- **id** is the primary key of the table;
- **artefact_1** is the ID of the (foreign key to the artefacts table);
- **artefact_2** is the ID of the second artefact (foreign key to the artefacts table);
- **distances** is an ARRAY of REAL values, containing the distances for each defined descriptor.

NOTE: Since the similarity matrix is symmetric, each pair of artefacts appears only once in the table. In order to enforce this, there is a uniqueness constraint on the pair of values (artefact_1, artefact_2)

The ordering of the similarity distances in the array is stored in the Python file:

`django-apps/backend/backend_defaults.py`

```
SIMILARITY_DESCRIPTORS = [
    "shell_bounding_box",
    "thickness",
```

[14] This may be useful to load the appropriate 3D model when loading a geometric property derived from it
[15] Further information at: https://docs.djangoproject.com/en/2.2/ref/models/fields/#choices

```
    "roughness_full",
    "roughness_skin",
    "skin_continuity",
    "colour_full",
    "colour_skin",
    "shape",
    "decoration_2d_floral_band",
    "decoration_2d_chequer",
    "decoration_3d_hatching",
    "decoration_3d_stamped_circles",
]
```

2.4 Similarity Search engine

The similarity search engine is a module of the API ENDPOINT that allows content-based retrieval on the artefacts of the collection. For each artefact, a series of descriptors are extracted. The descriptors of each artefact are then compared to each other to produce similarity distances (computed with the SimilarityLib software). The set of similarity distances for each descriptor is usually organized as a matrix (called "similarity matrix") and encodes the distances between each element of the set.

The algorithm [6,7] that performs the similarity query takes as input **a set of "activated" descriptors**, which are selected by the user, and the **query object**. For each descriptor, the algorithm filters the objects of the dataset with the similarity distance below a given threshold. Besides, a **relax** parameter has been introduced, which allows to broaden or restrict the result set, if the user considers the result set too small: it could also be empty with the default parameters.

The objects passing the threshold test for all the activated descriptors are returned in the output; besides, they are also ordered by score. This score is computed as the product of the similarity distances of the individual activated descriptors.

NOTE: currently, the thresholds are hard-coded in the API layer, and are applicable only to the Salamis and Naukratis datasets. This allows us to replicate the same results as the original GRAVITATE platform. These thresholds depend on both the dataset and the specific descriptor, since the values are not normalized. The thresholds are stored in the **`config.py`** module of the **`backend`** app.

The database schema for the similarity matrices of the search engine is represented in Figure 4.

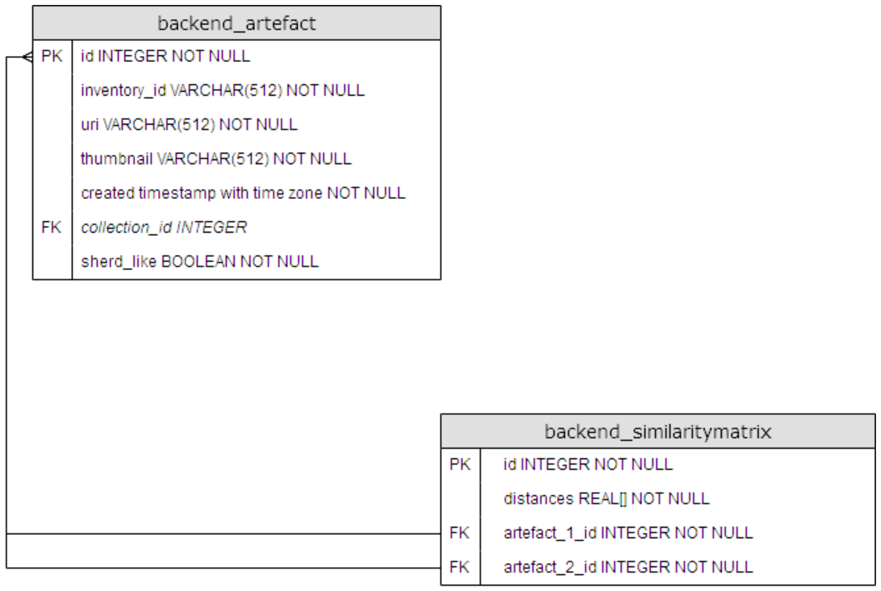

backend_artefact
PK id INTEGER NOT NULL
inventory_id VARCHAR(512) NOT NULL
uri VARCHAR(512) NOT NULL
thumbnail VARCHAR(512) NOT NULL
created timestamp with time zone NOT NULL
FK collection_id INTEGER
sherd_like BOOLEAN NOT NULL
backend_similaritymatrix
PK id INTEGER NOT NULL
distances REAL[] NOT NULL
FK artefact_1_id INTEGER NOT NULL
FK artefact_2_id INTEGER NOT NULL

Figure 4: Database schema for the similarity search engine; the artefacts table is the same as Figure 3

The table **similarity_distances** contains the similarity measures for all the available descriptors. The columns are described as follows:

- **artefact_1_id** and **artefact_2_id** are the IDs of the two artefacts. Since the matrix is symmetric, each pair of artefacts is entered only once in the table, to save space[16];
- **distances** is an array of real values, containing the distances for each descriptor; this field uses the ARRAY type.

A note about the usage of the ARRAY type: this type is not part of the SQL standard, so it could not be available in other RDBMS (such as, SQLite or MySQL).

The most standardized solution, which was also used for the past VISIONAIR[13] project, is to have two different columns containing a single value: one for the descriptor type, and one for the distance value. The main drawback of not using the array type, is that the size of the table is $N^2/2 \times D$, where N = number of artefacts, and D = number of descriptors: with this table, the number of rows increments of $N \times D$, for each added object. Thus, a more compact approach has been considered preferable here.

The usage the ARRAY type has several advantages:

- The representation is the most compact as possible: the number of records in the table is $N^2/2$;
- The implementation of the query functions is simplified, because the similarity_test function is always the same, regardless of the number of descriptors used;
- Adding a new descriptor can be done by simply appending a new value to the end of the array.

A disadvantage of this approach is that the table is not self-explanatory: the ordering of the descriptors is implicit and is determined by the import procedure of the **`import_similarity_matrices`** script, an administration script of the API endpoint.

This potential issue is mitigated by the usage of a global configuration parameter in the API level, called **`SIMILARITY_DESCRIPTORS`**, which determines the order of the descriptors in the table. Both the import_similarity_matrices and the similarity search API use the same configuration. This ensures that the same ordering is used for both INSERT and SELECT statements. The core of the search engine is a PL/PgSQL[17] function stored in the database, called **`similarity_query`**, with the following parameters:

[16]This means that the ID of the query item could be either in the artefact_1_id or the artefact_2_id column; both should be considered when looking for the query object
[17]https://www.postgresql.org/docs/current/plpgsql.html

- **`query_id`**: INTEGER,
- **`thresholds`**: ARRAY of REAL values.

This function filters the items in the table that have **`query_id`** equal either to **`artefact_1_id`** or **`artefact_2_id`**, and that the **`similarity_test`** function returns a **`True`** value.

The function **`similarity_test,`** which returns a Boolean value, has the following parameters:

- **`distances`**: ARRAY of REAL values,
- **`thresholds`**: ARRAY of REAL values.

The value of each i-th element of **`distances`** is compared with the corresponding entry of **`thresholds`**. If **`distances[i] < thresholds[i]`**, for all the values of **`i`**, then the object is considered "similar". The **`thresholds`** values depend on which descriptors are currently activated; in case a descriptor is deactivated, the threshold is set to **`-1`**. In this case, the loop inside the **`similarity_test`** function will skip the i-th descriptor.

The actual value of the threshold, for each descriptor, has been set manually to discriminate about the 5% of the whole dataset.

The similarity search engine is accessible from the API through the following endpoint:

`/api/artefacts/<id>/similarity_search`

where **`<id>`** is the ID of the artefact used as query object.

The endpoint expects a POST requests containing a JSON object with the following elements:

- descriptors: an array of strings, each entry is an activated descriptor,
- relax: the relax parameter.

Example of request:

```
POST /api/artefacts/4/similarity_search

{
  "descriptors":
  [
    "thickness", "roughness_skin", "colour_skin"
  ],
  "relax": 1.0
}
```

2.5 RESTful API

The database tables described in section 2.3 are mapped to "models" in the Django ORM[18] (Object-Relational Mapper). These models are classes that provide access to the underlying database, decoupled from the specific type of database (e.g. relational or key-value store). When used with a relational database, like PostgreSQL, operations on models are translated to the corresponding SQL statements (e.g. **`SELECT`**, **`UPDATE`**, **`DELETE`**).

To provide access to models to external clients (i.e. the desktop and the web client), it has been implemented an API layer based on the REST architectural style. A RESTful API can be implemented with little effort thanks to the Django REST Framework library, which extends Django by providing specialised classes to produce a full-featured RESTful API with few lines of code.

In this section, we give a brief description of the API endpoints; the full auto-generated documentation of the API can be found visiting the **`/api-docs`** path. Besides, when the application is in debug mode, the API can be navigated by accessing the main API endpoint (**`/api/`**) from a web browser. This will return a self-describing[19] HTML representation of the API.

The methods provided by each endpoint are pretty much the same:

Method	Endpoint	Description
`GET`	**`/api/<model-name>`**	List the items in the models
`GET`	**`/api/<model-name>?page=1`**	List the items in the model using pagination (currently, supported only by the **`/api/artefacts`** endpoint)
`GET`	**`/api/<model-name>/<id>`**	Retrieve a single instance of the model having primary key **`id`**
`POST`	**`/api/<model-name>`**	Create a new instance of the model; the primary key is auto-generated
`PATCH`	**`/api/<model-name>/<id>`**	Edit an instance of the model with primary key **`id`**
`DELETE`	**`/api/<model-name>/<id>`**	Delete an instance of the model with primary key **`id`**

The endpoints implementing each of these operations are the following:

Endpoint	Description
`/api/artefacts`	API endpoint for the Artefact model. It implements also a custom action **`similarity_search`** (see below)
`/api/groups`	API endpoint for the Group[20] model
`/api/geom-properties`	API endpoint for the GeometricProperty model
`/api/clipboard-item`	API to the ClipboardItem model
`/api/collections`	API to read/manipulate the ArtefactCollection model This is read-only, except for admin users

Besides these endpoints, there are others that do not follow this convention:

`GET`	**`/api/artefacts/<id>/similarity_search`**	Perform similarity search on the artefact with primary key id. Requires also a JSON payload

[18] https://docs.djangoproject.com/en/2.2/topics/db/
[19] https://www.django-rest-framework.org/topics/documenting-your-api/#self-describing-apis
[20] This is the Group model described in section

	(custom action)	containing additional parameters. See Section 2.4 for further information
`GET`	**`/api/chmetadata/<id>`** (read-only endpoint)	Retrieve the Cultural Heritage metadata associated with the artefact having primary key ID. This is a read-only endpoint, so it provides only the **`GET`** method. **NOTE**: the metadata are not available for all the fragments in the repository; in that case, an empty result set will be returned.

2.5.1 Retrieval of cultural heritage metadata

The original GRAVITATE platform offered also a Cultural Heritage metadata endpoint, accessible at the path **`/gravapi/metadata/ch`**. This endpoint takes as input the artefact URI, and provides the corresponding metadata as a JSON object. This endpoint has been implemented specifically to provide the Desktop Client with a JSON API to retrieve the Cultural Heritage metadata related to a specific artefact. In the GRAVITATE platform, this endpoint is used only for the Desktop Client, while the Web Client has already a dedicated functionality for cultural heritage metadata, inherited by ResearchSpace.

In the new IMATI architecture, which does not derive from ResearchSpace, another direction was taken: it was reused the Blazegraph journal from the original GRAVITATE platform, and the Cultural Heritage metadata endpoint was re-implemented, copying the SPARQL query from the original platform. The API endpoint in the new platform is **`/api/chmetadata`**, and is used both in the Desktop and the new Web client, which display exactly the same metadata.

NOTE: a known bug about the Cultural Heritage metadata API endpoint **`/api/chmetadata`** is that, for some artefacts, it returns an empty set. This is most likely due to the lack of a RDF property specified in the SPARQL query: as a result, the SPARQL engine returns an empty result set. Adding some **`OPTIONAL`** clauses in the query may solve this issue[21]. For this reason, in both the web and desktop clients, it may happen to visualize artefacts without the corresponding catalogue metadata.

2.6 Web-based client (front-end)

2.6.1 Overview

The code of the web-based client can be found in the the **`django-apps/webclient`** app. The content of the directory follows the default structure of Django apps[22]. Further information on the structure of Django apps can be found at the following link:

https://docs.djangoproject.com/en/2.2/ref/applications/

The main building blocks of Django apps are **Models** and **Views**. While models have been already described in section 2.5, we can say that views are classes that take care of rendering specific web pages. Views make use of templates, which are HTML files containing substitution variables, if-then-else statements and loops. The use of templates makes it really easy to have pages with the same style.

The web client app defines no models, since it makes use of the backend API endpoints. In future developments, however, it may be necessary to add some database table that is specific to the web client (for example, a table containing predefined settings of the web client UI). In this case, a corresponding Model class should be added to the **`webclient/models.py`** module, rather than the **`backend/models.py`** module.

The web interface is composed of the following pages:

Path	Description
`/`	Home page

[21] The query is found in the module: **`django-apps/backend/chmetadata_api.py`**

[22] Further information on the structure of Django apps: https://docs.djangoproject.com/en/2.2/ref/applications/

`/accounts/login`	Login page
`/browse/`	Browsing page
`/similarity-search/<id>`	Similarity search engine interface: shows artefacts similar to that specified in the ID parameter, according to a set of descriptors
`/view-artefact/<id>`	Shows details of an artefact
`/admin/`	Administration panel[23]

2.6.2 Similarity search interface

The similarity search interface provides a way to search items using geometric similarity criteria, as described in [6,7].

The user interface is similar to the one that was included in the original GRAVITATE platform; Figure 2 shows the "parameters" panel of the similarity search interface.

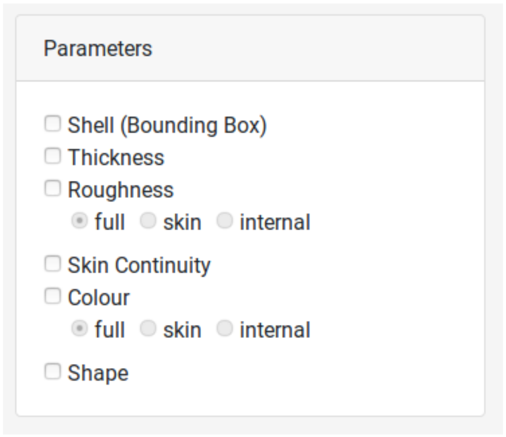


Figura 1: The "Parameters" panel in the similarity search UI

In order to make the interface easier to maintain, and also to provide a more pleasant user experience, this page makes use of a Javascript library called **Vue.js**[24]. Vue.js is a framework for building **reactive** user interfaces with HTML 5 and Javascript. "Reactive" means that the HTML user interface is automatically synchronized with the application state (stored in a Javascript object), making the development of rich UI components much easier.

The user interface is modularized thanks to **declarative components**, another essential concept of Vue.js. Declarative components can be seen as "custom HTML tags", combining HTML layout with Javascript application logic. Vue.js takes care of rendering these custom components properly, thanks to its templating engine.

One of the main advantages of Vue.js is that it can be easily integrated in an existing web framework like Django, making parts of the user interface more dynamic.

In the case of the Similarity Search interface, clicking the *Search* button will return the corresponding results, without reloading the page. This is true also for the *Find more results* and *Find fewer results* buttons, that respectively increase and decrease the *relax* parameters, and reload the results.

The reusable components are related to the "activated descriptors", and are part of the "search parameters" panel. The first is the **`property-selector`** component, which is composed of a checkbox and (optionally) a set of radio buttons. Each radio button is an option for which facet is considered, in the computation; the options are: full (full object), skin (skin facet) or internal (internal facet). However, the internal facet option is not available yet, since the facets have been extracted only for the "skin" part. The corresponding radio button has been disabled in the UI.

[23] There is no explicit link to the administration page in the web interface; however, it is accessible by visiting the path /admin/ of the host. For example, if the hostname is localhost, the admin page can be accessed by visiting: http://localhost/admin
[24] https://vuejs.org/

The other component is **decoration**, and is made of a thumbnail image and a checkbox. This component renders a thumbnail representing a decoration pattern, and the corresponding checkbox to activate the corresponding descriptor. Both components are defined in the **django-apps/webclient/static/webclient/vue/search-ui.js**

2.6.3 Administration panel

The administration panel can be accessed by visiting the **/admin/** page.

To access the administration panel, a super user account should be created first.

The command to create a new super user is:

```
$ docker-compose run --rm django-apps python manage.py createsuperuser
```

Or with the django-admin.sh script from the root directory:

```
$ ./django-admin.sh createsuperuser
```

Once the super user account has been created, it is possible to log in to the administration panel.

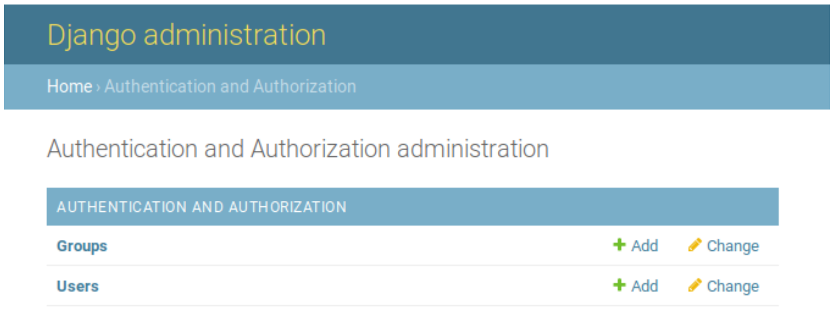
The administration panel allows to manipulate some of the existing models. By default, it allows editing of **Users** and **Groups** of users. Other models of the back-end APIs have been included as well: they are **Artefacts** and **Collections**.

To create a new user, log in to the administration panel by visiting the

/admin

page with a super user account.

To add new users, select the "+ Add" link near to "Users". The new users created this way will have by default no administration rights.


Django administration
Home › Authentication and Authorization
Authentication and Authorization administration
AUTHENTICATION AND AUTHORIZATION
Groups
+ Add
Change
Users
+ Add
Change

2.6.4 Blazegraph administration panel

Besides the Django built-in administration panel, there is also the Blazegraph administration panel, which can be accessed visiting the **/blazegraph** path. Only users with administration rights have access to it.

For simplicity of set-up, the access control is obtained using the Django server as a reverse proxy, so this panel will be accessible only if the **django-apps** service is running.

Through this panel, it is possible to do various operations on the triple store, like importing new data, updating the existing ones, and testing SPARQL queries.

2.7 How to and FAQs

2.7.1 Adding a new feature descriptor in the similarity matrix

To add a new descriptor in the similarity search engine:

1. In the **django-apps/backend/backend_defaults.py** module add a new entry to the **SIMILARITY_DESCRIPTORS** string array, corresponding to the descriptor you want to add;
2. In the **datasets/datasets.json** file, add a key-value pair under the "matrices" attribute, for each dataset, where the key is the same string entered in step 1, and the value is the relative path to a .txt file containing the similarity matrix of a specific dataset;

3. Run the command:
   `./django-admin.sh import_similarity_matrices` from the root directory. This will regenerate completely the similarity matrix, but the new descriptor will be added in the position determined by the **`SIMILARITY_DESCRIPTORS`** array.

Up to this point, the new descriptor is in the database and can be used when calling the similarity search API (see section 2.5). To add a corresponding GUI control, you also need to edit the file:

```
django-apps/webclient/templates/webclient/similarity-search.html
```

At the end of this file, there is a **`<script>`** tag where an object of **`Vue`** type has been declared. This object declares the dynamic behaviour of the DOM element handled by *Vue.js*.

In the data attribute, there is a dictionary of key-value pairs called **`info`**. To add the controls of the new descriptor:

- Add a new entry to the **`info`** attribute (a dictionary of key-value pairs) using as key the new entry added in the **`backend_defaults.py`** module
- Add the same string to one of the arrays in the **`sections`** object (if it is not a descriptor related to pattern recognition, its place is probably in the **`base`** array).

2.7.2 Reset the database completely

Open a terminal in the root directory.

To stop all the running containers run:

```
$ docker-compose down
```

Then, remove the existing database files:

```
$ sudo rm -rf pgdata
```

Finally, regenerate the database:

```
$ ./init_and_populate
```

NOTE: this procedure will delete the content of all tables, including the **`django_user`** table, the **`backend_group`** and **`backend_clipboarditem`** table; if you want to maintain the content of these tables, you need to make a dump of the database beforehand.

3 Desktop client

This section is intended as a "map" of the codebase, made to facilitate the task of extending and modifying the code.

The project produces as output two executables: "*gravitate-client*" (the main desktop client) and "*standalone-mesh-viewer*" (an off-line visualizer for meshes that is able to perform part annotation). These executables depend on a common library, composed of the code in the "*common*" directory.

This document is organized as follows: Subsection 3.1 describes how the whole codebase is organized. Then, Subsection 3.2 describes the code of the Desktop Client; Subsection 3.3 describes the code of the Standalone Mesh Viewer; Subsection 3.4 describes the common library. Finally, Subsection 3.5 contains notes on how to package and deploy the code to the final user.

The instructions to compile the code can be found in the **INSTALL.md** file in the main directory of the project.

3.1 Project structure

3.1.1 Root directory

The codebase is mainly composed of three parts:

- **Desktop client code**: the code that belongs to the **`gravitate-client`** executable; it is found in the **apps/desktop-client** directory, and depends on the "common" library;
- **Standalone Mesh Viewer**: the code that belongs to the **standalone-mesh-viewer** executable; it is found in the **apps/standalone-mesh-viewer** directory, and depends on the "common" library;
- **"Common" library**: a library of common functionalities, mainly for the handling of 3D models and their properties; both the executables mentioned above are linked to parts of this library.

There are other additional directories, which mainly support the build/deployment process, but do not contain actual application code:

- **cmake**: it contains the **`Config.cmake`** file, which is the main configuration file for the project;
- **deployment**: this directory contains deployment scripts, which allow to create distributable versions of the desktop client, including all the required third-party libraries;
- **docs**: the directory that contains general documentation of the project, and the integrated help files;
- **resources**: it contains all the resources of the project (images, text files) that are accessed throughout the code. Using the Qt resource mechanism, these files are compiled directly in the final executable;
- **tests**: an automated test suite written with the Google Test library;

thirdparty: it contains CMake scripts that download and compile some external dependencies:

- **JSON For Modern C++**: a JSON serialization library for C++;
- **Qxt**: a utility library that extends the base Qt library;
- **Serd** and **Sord**: RDF parsing libraries;
- **fast-cpp-csv-parser**: a library to parse CSV files.

3.2 "Desktop Client" application

This section describes the code structure of the desktop client application. The sub-section 3.2.1 *Client-server architecture* describes how the client communicates the back-end server, and how it is possible to support different back-ends. The sub-section 0 -

Organization of the directories provides a description of the directories composing the desktop client code. For each directory, a brief explanation of the contained classes is given.

3.2.1 Client-server architecture

Overview

The GRAVITATE platform is built with a client-server architecture; the Desktop Client communicates with the server using a RESTful API.

Currently, the desktop client is compatible with two different back-ends:

- The GRAVITATE back-end, which was developed for the GRAVITATE project, and is not constantly maintained anymore;
- The IMATI back-end, which was developed afterwards as a replacement, to consolidate the development work done by IMATI during the project, and to prepare it for future applications and extensions.

The access to the back-end is provided through the Repository classes, which encapsulate the calls to the back-end API, according to the Repository design pattern. Operations on a Repository class are defined on an abstract class: this class has a concrete implementation for each of the back-ends.

For example, the **`Metadata3DRepository`** class (the repository for 3D metadata) has two implementations: **`GravitateMetadata3DRepository`** and **`ImatiMetadata3DRepository`**.

The use of this pattern makes the rest of the client code independent of the back-end that is actually used.

The choice of the back-end is done at run-time, according to the **`backend`** property of the **`settings.ini`** file:

```
[General]
backend=GRAVITATE
hostname=http://gravitate-2.cytera.cyi.ac.cy
user=<USER NAME HERE>
password=<PASSWORD HERE>

[scheduler]
user=<SCHEDULER_USER>
password=<SCHEDULER_PASSWORD>
```

If the **`backend`** property has value **`GRAVITATE`**, the GRAVITATE back-end is used; if the value is **`IMATI`**, the IMATI back-end will be used instead.

Since the Repository classes form a family of related classes, they are instantiated through an Abstract Factory, which chooses the right implementation according to the application settings. The abstract factory is the class **`BackendFactoryBase`**, and its concrete implementations are **`BackendFactoryGravitate`** and **`BackendFactoryImati`**.

NOTE: The **`settings.ini`** file contains two sets of credentials, one in the "general" section and one in the **`[scheduler]`** section. The scheduler credentials are used for the **Job Scheduler** service. Further information about the Job Scheduler can be found in the corresponding section of the documentation.

ID classes

In the **`core`** directory there are some "identifier classes": **`ArtefactId`**, **`GroupId`** and **`ContainerId`**, which represent the identifiers of artefacts, groups and containers.

These classes are wrappers around the concrete type used for the IDs: this helps validity checks and avoiding accidental comparisons between different ID types (e.g. comparing an Artefact ID with a Group ID will cause a compilation error).

The concrete type differs according to the back-end used: for the GRAVITATE back-end is a URI type (**`QUrl`** class), while for the IMATI back-end is an integer type (**`int`** primitive type).

When using the ID classes, it is therefore important not to mix different types: **this responsibility is left to the developer.** The reason for the usage of integer IDs is due to how the access to the database is handled in Django. First, Django uses an ORM (Object Relational Mapper) which maps Python objects to corresponding database entities. For each of these entities, a class deriving from **`django.db.models.Model`** has been declared. In relational databases, models map with corresponding tables, and by default, the primary key of these tables is of integer type. So, for simplicity of implementation, it has been considered better to follow this pattern.

However, to maintain the compatibility with the original GRAVITATE platform, the artefact URI has been included in the database table, and it is possible to retrieve an object by URI rather than by primary key.

Clipboard

The Clipboard is one of the main components of the user interface, and in the GRAVITATE architecture is also the main synchronization mechanism between the Desktop and the Web clients. The clipboard is a space where the user is able to store artefacts for later usage during the workflow.

In the new IMATI platform, the role of the Clipboard is limited to the Desktop client: with the browsing functionality it is possible to navigate the repository and add new items to the clipboard of the user, whereas the GRAVITATE web client provided a corresponding view on the Clipboard, thus having a full round-trip between the two clients. Consequently, the Web Client and the Desktop Client do not communicate in the current implementation. In order to have a full round-trip also in the IMATI platform, only, is the GUI component has to be implemented: in particular, a Clipboard widget on the web client is missing, since the API methods to read and manipulate the Clipboard have been already implemented.

The class offering access to the Clipboard on the back-end is the **`ClipboardModel`** class. This is an abstract class, derived from **`QAbstractItemModel`**, which defines access to the clipboard. Through this class, it is possible to retrieve the current state of the clipboard, and also to manipulate it (create new groups, adding items to existing groups, removing items).

As for the other classes dependent on the back-end, it has two subclasses with the back-end specific code: **`GravitateClipboardModel`** and **`ImatiClipboardModel`**. The appropriate subclass is chosen at run-time using a factory method (as described in the "Overview" sub-section).

Items in the Clipboard can be added either as unorganized items, or as part of the group; in the latter case, deleting a group will also delete all the objects inside.

The abstract methods that have a back-end specific implementation are the following:

- **`loadClipboard`:** loads the clipboard content from the back-end,
- **`createGroup`:** creates a new group,
- **`addToGroup`:** adds an artefact to an existing group,
- **`addToClipboard`:** adds an artefact as an unorganized item,
- **`removeResource`:** remove a group or an individual item[25].

`ClipboardModel` is a Model class, according to the **Qt Model/View**[26] architecture: thus, it can have a GUI representation by simply instantiating a **`QTreeView`** and calling its **`setModel`** method, with a **`ClipboardModel`** instance as parameter. In practice, the **`QTreeView`** is used as part of another widget class (**`ClipboardPanel`**, see sub-section *"clipboard" directory* for further information), which adds a special handling of the contextual menu actions.

Repositories

The back-end services are accessed through objects that follow the Repository[27] pattern.

Repository classes have methods to retrieve and, in some cases, update existing data. This helps maintainability because the query, retrieve and update operations are called by abstract interfaces. Repository classes are defined in the **`backend`** directory (see section *"backend" directory* for further information).

Instantiation of the repository classes does not happen directly with calls to constructors, but is mediated with a factory method of the **BackendFactoryBase** class (there is one for each repository class). This ensures that the right subclass is chosen, according to the current configuration of the **`settings.ini`** file.

There is a series of Repository classes that have been defined:

[25] Depending on the argument passed, which can be either a group or an individual clipboard item
[26] https://doc.qt.io/qt-5/model-view-programming.html
[27] https://deviq.com/repository-pattern/

- **MetadataChRepository**: returns the Cultural Heritage metadata of an artefact with a given **ArtefactId** (see Sec. 2.5.1);
- **Metadata3DRepository**: returns the 3D metadata of an artefact with a given **ArtefactId**;
- **ArtefactListRepository**: returns a sequential list of items, which is used to implement the *Browsing* functionality; it also supports pagination of the results (i.e. displaying the results split in pages of equal size rather than a long list, thus reducing load times and positively affecting the user experience);
- **LoginProvider**: perform login to the system, by making a request to a login endpoint. The GRAVITATE back-end lacks a dedicated login end-point: the login endpoint is replaced by a call to the Clipboard API, which returns a successful HTTP 200 response only if the credentials are correct;
- **AbstractThumbnailRepository**: given the artefact ID of a fragment, it returns the corresponding thumbnail.

Besides the Repository classes, the **backend** directory contains also:

- the **ClipboardModel** class (see sub-section "Clipboard");
- the **AbstractDownloader**, a utility class built as a wrapper around **QNetworkAccessManager**. It performs download of files, with caching support for large 3D files. This caching mechanism is explained in detail in the section Local cache for downloaded files;
- The **JobManagerBase** class that performs job requests, and at fixed intervals polls the state of the current job, to verify whether it has been completed. It is described in detail in the Job Scheduler section.

NOTE: in the GRAVITATE back-end, there is also a **GravitateExtraMetadata** class. Its purpose is to add some off-line metadata, which could not be included in the back-end, but that it was required to be displayed for the users for demo purposes. Currently, this class is used to provide the average thickness property. The average thickness is read from a CSV file, and covers only the Salamis dataset.

Networking

The network requests are performed with the class **NetworkBackendProvider**, which implements methods from the **BackendProvider** interface. The **BackendProvider** class provides access to the API endpoints, and is configured according to the settings in the **settings.ini** file.

The authentication is HTTP Basic because it is the only authentication mechanism offered by the GRAVITATE backend. Other authentication mechanisms could be implemented, like a token based authentication based on JSON Web Tokens, a very popular format which is supported by many back-end frameworks.

Local cache for downloaded files

The class **LocalFileCache** provides a caching mechanism for the files downloaded from the back-end. Although the **QNetworkAccessManager** can incorporate a caching mechanism, this is a secondary cache that replicates the same directory structure of the files on the server.

Job scheduler

For backwards compatibility, the client-side part of the job scheduler code has been kept in the codebase, although it is used only for the GRAVITATE back-end. If implementing the corresponding functionality for the IMATI back-end, it would be possible to reuse the common parts (i.e. deriving a new class from **JobManagerBase**)

The **[scheduler]** section of the **settings.ini** file contains a secondary set of username-password credentials. These are used that perform requests to the **/gravitate-scheduler/jobs** endpoint, which is the job scheduler.

The scheduler is used to execute specific processing and analysis algorithms on the back-end, like the Feature Extraction (developed by IMATI) and the ReAssembly algorithm (developed by Technion/Haifa). Of these algorithms, only the Feature Extraction has been successfully integrated.

The execution of a job is performed as follows:

1. It is called the **startJob** method of the **JobManager** class. The method takes two parameters: the **Algorithm ID** (e.g. in the case of the Feature Extraction algorithm, is **featuresExtraction**) and a payload string, containing the parameters of the algorithm (each algorithm defines its own format for the input parameters);

2. The call to the **startJob** method returns a Job ID. The **GravitateJobWatcher** class polls the **/gravitate-scheduler/<JOB_ID>/result** endpoint until the job returns a valid result, or an error message if the execution has failed;
3. The client application consumes the results of the algorithm computation, such as visualizing the features extracted from the 3D model.

NOTE: The "base" job scheduler is implemented in the class **JobManagerBase**, which is an abstract class. The abstract methods are **startJob** and **cancelJob**, and they have a concrete implementation in backend-specific subclasses (**GravitateJobManager** and **ImatiJobManager**). Note that in the case of **ImatiJobManager**, the **startJob** and **cancelJob** methods are empty, because a corresponding **Job Scheduler** component, responsible for the algorithm execution, has not been implemented (see Section 2.1).

3.2.2 Organization of the directories

This sub-section describes the directories of the "desktop-client" application, one by one.

"backend" directory

This directory contains the back-end related code. The subdirectories are organized as follows:

- **base**: contains base abstract classes,
- **gravitate**: contains the classes related to the GRAVITATE back-end,
- **imati**: contains the classes related to the IMATI back-end,
- **network**: contains classes related to network-specific functionalities (managing HTTP requests).

"browse" directory

This directory contains the code pertaining to the *Browsing* functionality (accessible from the Data → Browse button).

The main class of this module is **BrowseDialog**, which implements a basic browsing dialog. This dialog can be accessed from the menu Data/Browse. It also contains a helper class called **DatasetFetchModel**, derived by **QAbstractTableModel**: the purpose of this class is to retrieve the data used by the **BrowseDialog**.

"clipboard" directory

The Clipboard is the place where the user can store artefacts for later usage in a workflow. The access to the Clipboard happens through a model class (**ClipboardModel**, see paragraph ""backend" directory", sub-section 0).

The GUI for the Clipboard is offered by the **ClipboardPanel** class. The **ClipboardPanel** is basically a wrapper around a **QTreeView** widget, with some additional signal-slot connections for handling the context menu appropriately.

The **onContextMenuRequested** private slot handles the context menu invocation, emitting a **contextMenuRequested** signal. The **contextMenuRequested** signal has the additional parameter of the resource currently selected.

This helps creating contextual menus at run-time, with different entries according to the active view.

"common" directory

This directory contains classes that do not belong to a specific category:

- The **AbstractDashboard** class is an abstract class for some of the "services" that are accessed by the views, like the job manager and the help manual;
- The **ArtefactModel** which manages all the data related to the artefact;
- The **ArtefactPresenter** class which is a mediator class between the **ArtefactModel** and the **InspectionItemWidget**.

"dialogs" directory

This directory contains some dialog windows that are used in different parts of the application:

- The **LoginDialog** is the main login dialog window;

- The **`ParameterSelectionDialog`** is a configurable dialog window which can be configured to accept a series of numeric or enumeration parameters; currently it is used for the parameters of the Feature Extraction Algorithm;
- The **`TextNoteEdit`** dialog is used in the Groups View, to add a note to a group or to the Clipboard. The GUI is just for demo purposes, as it is not possible to save the note in the back-end.

"help" directory

This directory contains classes related to the **online help** functionality.

The online help consists of two functionalities: the "What's this" text which is a brief descriptive sentence about a button, and the help manual.

- The **`HelpDialog`** class is a dialog widget which is invoked when the user presses the F1 button, or from the Help → User Manual menu;
- The **`HelpBrowser`** retrieves the pages of the help manual, and presents them inside the **`HelpDialog`**;
- The **`StringResources`** class is a parser for the Android String Resources XML-based format; this way, all the string resources can be stored in a XML file and retrieved by their name;
- The **`HelpHandler`** retrieves the string constants of the **`helpstrings.xml`** file and returns them using the **`getText`** method. These strings are used as "What's this" text.

Help manual: notes on the automatic generation

The help manual uses the Qt Help Framework to organize and package the manual pages.

The help manual is written in Markdown format, and thanks to the Doxygen tool is translated in HTML, then compiled as a Qt Help Collection File (see the Qt Help Framework page for further details).

To compile correctly the Qt Help Collection, it is necessary to have installed the following tools:

```
$ sudo apt install doxygen texlive texlive-binaries graphviz
```

If any of the required tool is missing, a compilation error will be given.

"metadata" directory

The main classes of this module are:

- **`Artefact3DMetadata`**: represents the 3D metadata of a specific artefact;
- **`MetadataModel`**: represents the Cultural Heritage metadata of a specific artefact.

The MetadataModel inherits from **`QStandardItemModel`**, and builds a hierarchical item structure where there are two predefined sections: one for the artefact provenance information, and one for the acquisition information of the 3D model file.

`artefacturladapter.h` contains a template function:

```
template<class T>

QUrl getUrl(ModelResolution res, const Artefact3DMetadata &artefact)
```

with template specializations for each geometric property: this way, each specialization of the **`getUrl`** function returns the appropriate URL at the required resolution, retrieved from the Artefact3DMetadata object.

`stringmanipulation.h` contains utility functions for string manipulation.

"rendering" directory

This directory contains classes related to rendering aspects that are specific of the Desktop Client. Its only class is **`DashboardSDFPropertyRendering`**, a specialization of **`SDFPropertyRendering`** which displays the average thickness of the fragments in overlay, in addition to the Shape Diameter Function.

"repository" directory

This directory contains properties that are specific to how the GRAVITATE repository is structured, in particular the relationship between the 3D models and the property files.

"widgets" directory

This directory contains widgets used in various parts of the main user interface:

- **`MeshPropertyToolbar`**: a container toolbar for selecting geometric properties, facets, minimal bounding box and annotations;
- **`PropertySelector`**: a specialized **`QComboBox`** for parameters of type **`ScalarFieldType;`**
- **`ViewSelectionBar`**: the upper toolbar to choose the current view.

"views" directory

The views directory contains the code that implements the views; currently there are five views:

- Inspection
- Fragment
- ReAssembly
- History
- Groups

This section contains an overview of the Model-View-Presenter architecture, followed by a description of the "views" directory content.

For a description of the views from the design perspective, please refer to the D4.1[1] and D4.2[2] deliverables of the GRAVITATE project.

Model-View-Presenter architecture

A GUI application, and in general any application that allows user interaction, is made up of code that may belong to either one of the following categories:

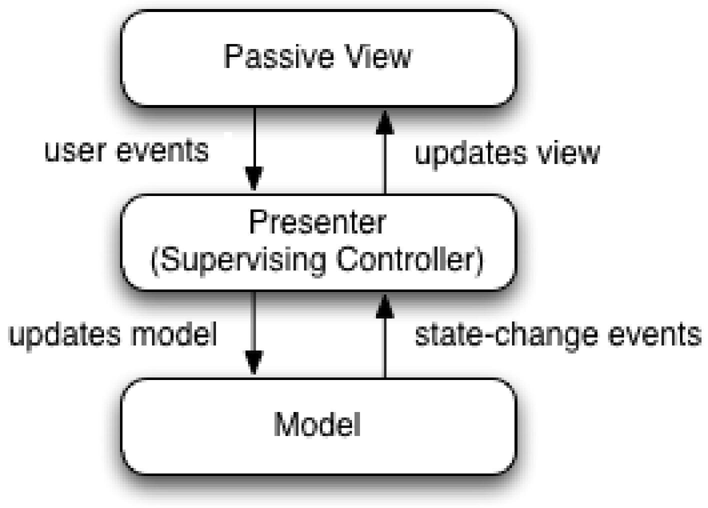
- *Business logic*: code that manages the creation, update, and storage of data;
- *Presentation logic*: code that handles the interaction with the user.

Ideally, a change to the business logic should not require a change to the presentation logic, and vice versa. This concept is called *loose coupling*, and is characterised by the fact that each part of the system should have little knowledge as possible of the other parts, so that they can be changed independently; conversely, when a change must be propagated throughout many (and conceptually independent) parts of the system, we have a *tight coupling*. A tightly coupled system is hard to maintain, and usually it should be avoided.

The Qt library already follows this principle, by encouraging Model-View programming; however in the long run, this approach did not reveal sufficient.

Thus, it has been considered useful to introduce an additional layer of indirection that manages the interaction between the business logic and the presentation logic. In particular, it has been chosen the Model-View-Presenter (MVP) architectural pattern, which is a variation of the Model-View-Controller (MVC) pattern[28].

[28] https://en.wikipedia.org/wiki/Model–view–controller



In the MVP pattern the presenter has the role of a "supervising controller", while the model is just an interface to the underlying data, and the view completely passive. This choice is convenient because of the specific nature of the Desktop Client application, where there is not always a clear distinction between what is the presentation logic and what is the business logic. So the optimal choice was a pattern where the intermediate layer contains most of the business and presentation logic, and mediates between the view and the model.

In the desktop client, for each view, there are three classes that correspond to the three components of the MVP pattern, with a distinctive suffix.

For example, for the Inspection View has:

- **`InspectionViewModel`** class (Model)
- **`InspectionViewPresenter`** class (Presenter)
- **`InspectionViewMain`** class (View)

NOTE: The View class has the "Main" suffix, because "View" is already part of the class name.

Views make extensive use of the Qt signals feature: each user input action produces a signal in the view, that the Presenter is able to subscribe to. The advantages of this approach are the following:

- The View is completely decoupled from the Presenter (i.e., the View does not need a pointer to the Presenter class) and the Presenter methods can have a **`private`** scope.
- The View signals represent the user actions in an abstract way; signals can pass explicit parameters to the Presenter, so that the corresponding action can be processed with a smaller effort. Alternatively, the View should have provided query functions, in case the parameters were required by the Presenter.

Inspection view ('inspection' directory)

This directory contains the Inspection View classes. Besides the MVP classes explained in the previous section, this directory contains the **`InspectionItemWidget`** class. This class contains all the basic features of the "rich" visualization of an artefact, in particular:

- A panel containing the 3D model viewport (**`MeshViewWidget`** class) with many default controls (light settings, length measurement, 3D model selection controls and others);
- A Cultural Heritage metadata visualization panel (a **`QTreeView`** connected with a **`ChMetadataModel`**);
- A part annotation list, based on **`QAbstractAnnotationModel`**. It contains controls to add/remove SKOS terms associated with a part, or to delete an annotation completely.

Fragment View ('fragment' directory)

This directory contains the Fragment View classes.

The fragment view has more manipulation functionalities; in particular, it enables the use of the part-based annotation and of the Feature Extraction algorithm.

Some notes about the classes in this directory:

- The `FragmentItemWidget` class is an extension of the `InspectionItemWidget` with additional functions for part-based annotation;
- The `FeatureExtractionAlgorithm` class prepares and executes the calls to the FeatureExtraction algorithm, which currently happens only on the GRAVITATE back-end. Note that in the Fragment View there is also the possibility to show a demo of the Feature Extraction results on a selected set of models.

ReAssembly View ('reassembly' directory)

The ReAssembly view, at the current implementation state, contains only an off-line demo of the ReAssembly functionality on a sample test case.

History View ('history' directory)

The History View has been designed as a utility view, to store documentation about the workflow that has been carried out.

Currently, it just a GUI without functionality, because the corresponding back-end was not implemented during the project.

Groups View ('groups' directory)

This view focuses on manipulating the clipboard and its object: adding/removing objects, and creating new groups.

Note: the button "Add note" opens a dialog (class `TextNoteEdit`) for creating a new text note. This GUI has been included just for demo purposes, as the back-end does not have a corresponding saving functionality.

3.3 Structure of the Standalone Mesh Viewer

The Standalone Mesh Viewer is a 3D model viewer, implemented with the same widgets of the Desktop Client.

This user interface contains the following functionalities:

- Loading a 3D model;
- Loading the geometric properties associated with it;
- Performing part annotation on the 3D model, with the same functionalities of the Desktop Client (Sections 3.4.1 and 3.4.2 describe the involved classes);

The GUI, as for the main desktop client, is organized according to the MVP pattern:

- **StandaloneMeshViewer**: view class
- **StandaloneMeshViewerModel**: model class
- **StandaloneMeshViewerPresenter**: presenter class

These classes are pretty self-explanatory; the remainder of the code is from the “common” library, described in Section 3.4.

3.4 Structure of the "Common" library

The common library is organized in the following directories:

- **annotation**: classes related to the part-annotation functionality
- **core**: units of measure and basic enumeration types
- **input**: classes that read/write in various input formats
- **meshvis**: classes related to mesh visualization and processing
- **util**: utility classes (math, time/date and others)

The following subsections describe the functionalities provided by the library, not following the directory structure, but by topic.

Most of the code, however, is included in the "mesh" directory, which is described in sub-section 3.4.1 - *3D visualization ("meshvis" directory)*, and in the "annotation" directory, which is described in section 3.4.2 - *Annotation-related classes*.

3.4.1 3D visualization ("meshvis" directory)

Overview of the 3D Model visualization

The widget that offers 3D visualization is the **`MeshViewWidget`**, which is a wrapper around the **`QOpenGLVtkWidget`** class of the VTK library.

The **`MeshViewWidget`** offers a pre-configured VTK context. The **`MeshViewWidget`** is a container for **`Mesh`** objects. Each mesh object contains the reference to a **`VTKPolyData`** object, which is the data structure used by VTK to store meshes.

Meshes can be added and removed to the **`MeshViewWidget`**, using the method **`addMesh`**. This way, it is possible to have more than one mesh in the same viewport (e.g. in the case of the ReAssembly View); however, typically each viewport contains a single file.

The **`MeshViewWidget`** is a container for "modules", which provide specific functionalities. The remainder of this section contains an overview of the currently available modules.

The **`input`** directory contains parsers for various file formats (facets, geometric properties, minimal bounding box, features and others).

Mesh visualization

3D models are loaded using the **`Mesh`** class. The Mesh class has a **`load3DModel`** method, which takes the path of the 3D model on the local disk. For 3D models downloaded from the back-end, the **`FileDownloadHelper`** class is used.

Once the 3D model has been loaded, it can be added to the **`MeshViewWidget`** using the **`addMesh`** method.

NOTE: in the Desktop Client, the **`MeshViewWidget`** is not used directly in the Inspection view and the Fragment View: indeed, it is wrapped in the **`InspectionItemWidget`** class, which is a widget that combines the 3D viewport and the cultural heritage metadata view; the **`InspectionItemWidget`** accepts a single 3D model as input, with the **`setMesh`** method. The metadata can be displayed using the **`setMetadataChModel`** method.

MeshViewWidget modules

These modules provide various functionalities related to meshes, from the visualization of various geometric properties to the selection and annotation of parts: they are found in the "common/mesh" subdirectory.

Default modules

There are some modules that are included by default in the Desktop Client user interface:

- **`CoordinateAxes`**: displays the XYZ axes in overlay,
- **`MeasureTool`**: tool for measuring distances on the mesh surface. Internally, it makes use of **`vtkCellLocator`** to improve performance,
- **`ValuePicker`**: if a geometric property has been loaded on a mesh, it is possible to see the current value by hovering the mouse on it.

The **`MeasureTool`** and **`ValuePicker`** modules add a corresponding button to the toolbar.

NOTE: the **`ValuePicker`** works only with scalar fields defined on vertices (Mean Curvature, Shape Index, Lightness) and not for those defined on triangles (Shape Diameter Function).

Geometric Properties visualization

The visualization of geometric properties is made with the class **`PropertyColoring`**. This module provides a generic visualization of scalar fields using a colour transfer function.

The association between a geometric property and its colour transfer function is done in the subclasses of **`AbstractPropertyRendering`**. This class has an abstract **`createLut`** method, which provides a lookup table between a property value and the corresponding colour.

The concrete subclasses of this method are **`MeanCurvaturePropertyRendering`**, **`ShapeIndexPropertyRendering`**, **`SDFPropertyRendering`** and **`LightnessPropertyRendering`**.

By subclassing **`AbstractPropertyRendering`** it is possible to add support for other geometric properties.

Facet visualization

Facets are loaded as **`Facet`** objects using the **`FacetReader`** class. The **`FacetOverlay`** class performs the visualization of facets by extracting parts of the original mesh and visualizing them on top of the original mesh.

Minimal Bounding Box visualization

The bounding box visualization is performed by loading the bounding box coordinates using the **`BoundingBox`** class. Once the bounding box has been loaded, it can be displayed using the **`WireframeBox`** class.

Part-based annotation

The core classes of the part-based annotation are the **`SelectionTool`** class and the **`AnnotationTool`** class.

`SelectionTool` is the class that performs the selection of the parts: it works on points, lines and surfaces.

`AnnotationTool` provides a visualization and manipulation functionality for the saved annotations.

Besides these classes, there is an **`AnnotationDirector`** class that coordinates the whole annotation process, making use both of **`SelectionTool`** and **`AnnotationTool`**.

Visualization of extracted features

The computation of features is made by calling the feature extraction algorithm on the back-end. Once the result has been returned to the client, a **`std::vector<HoughFeature>`** is created using the **`HoughFeatureReader`** class. The **`HoughFeatureTool`** class visualizes the feature vector in the GUI.

Units of measure

Scans of the 3D models are produced in a given unit of measurement. For achieving compatibility between models scanned in different units of measurement, it has been implemented a conversion utility, based on the QUDT ontology. This utility is found in the **`qudt_units.h`** file. This header file is automatically generated from the QUDT ontology.

The script **`generate_qudt_units.py`** extracts all the units of measure from the QUDT vocabulary, and produces a **`Unit`** enum, made of all the linear, square and volumetric units provided in QUDT.

The function **`fromUri(const std::string &uri)`** provides a conversion from the URI of each unit of measure in the QUDT vocabulary, and its corresponding Unit value.

3.4.2 Annotation-related classes

The “common/annotation” directory contains some classes that are related to the part-based annotation, but not to the actual 3D manipulation.

- The **`ThesaurusModel`** class is a model class that provides navigation of the vocabulary. The vocabulary can be loaded from a RDF document containing SKOS terms (i.e. instances of the SKOS ontology);
- The **`PartonomyWidget`** is a widget that uses ThesaurusModel to display and navigate the tree structure;
- The **`AnnotationModelBase`** class, which is an abstract class used to provide access to the part annotations of a specific 3D model, and to make annotations persistent; the part of the class that is left abstract is that related to persistence.

NOTE: **`AnnotationModelBase`** has a concrete implementation, **`LocalAnnotationModel`**, which saves the annotations on the local disk.

Currently, a full integration of the annotation functionality in the back-end of the platform is not available. Such integration can be implemented, by creating another subclass of **`AnnotationModelBase`** (e.g. **`BackendAnnotationModel`**). The **`BackendAnnotationModel`** class would provide an implementation of the abstract methods of **`AnnotationModelBase`**, so that files are read and written on the annotation endpoint, and not on the local disk, as it is now.

The easiest (and recommended) way is to do it is to subclass **`AnnotationModelBase`** appropriately, implementing the corresponding abstract methods defined in the class, so that the Annotation Model class loads and stores annotations on the annotation endpoint.

3.5 Deployment of the code

Once the GRAVITATE client has been successfully compiled, it is possible to generate a distributable package containing all the required dependencies.

NOTE: the scripts in the **`deployment`** directory in the source code are just templates for the actual deploy scripts, that you can find in the **`deployment`** directory of the "build" directory tree (hence the **`.in`** suffix).

3.5.1 Linux

The distributable package format for Linux is the AppImage format, using the *linuxdeployqt* tool.

```
$ ./deploy-linux.sh
```

The *linuxdeployqt* tool produces AppImage files that are compatible with all the supported linux distributions.

The package will be generated on the path **`dist/linux/gravitate-client-x86-64.AppImage`**

The AppImage can be run with:

```
$ ./gravitate-client-x86-64.AppImage
```

No root privileges are required.

NOTE: the **`dist/linux`** directory contains also the uncompressed files that compose the distributable package.

NOTE: Because of the way **`linuxdeployqt`** works, the creation of the distributable package is expected to be made on an "old enough" version of the operating system. The reason is that the libraries included in the package are linked against some system libraries (e.g. **`libc.so.6`**) which are not included in the package, so they are expected to be found on the host system when the package is executed. Compiling the code on a "too new" version of the operating system may cause errors if the package is executed on an older distribution, while it would not cause any problem to execute it on a newer version of the operating system.

The **`linuxdeployqt`** tool handles this situation by explicitly preventing the compilation explicitly on a "too new" version of the operating system: indeed, the authors of the tool recommend to use the oldest supported distribution, to maximise compatibility (at the moment of writing, Ubuntu 16.04 – Xenial Xerus). If a newer distribution is used to produce the distributable package, an error will be given.

Further information can be found at the following links:

- https://docs.appimage.org/introduction/concepts.html#build-on-old-systems-run-on-newer-systems
- https://github.com/probonopd/linuxdeployqt/issues/340

3.5.2 Windows

The distributable package is produced by opening a command prompt (**`cmd.exe`**) and entering:

```
> deploy-windows
```

The **`deploy-windows.cmd`** script will collect the files and put them in a directory called **`gravitate-client-<DATE>`**, where **`<DATE>`** is the current date. This directory can be compressed and distributed as a stand-alone executable.

4 Guidelines for the implementation of the Annotation API

4.1 Overview

This section provides some guidelines on a possible implementation of the annotation back-end functionality, which currently is a future work.

In the original GRAVITATE architecture design, the back-end was expected to have an annotation server, which was responsible for receiving annotations made by users of the platform. This way, annotations could be shared with other users, and could also be indexed by the semantic search engine, so that they would appear as results of complex queries combining geometry and semantics. However, the implementation of this server did not happen during the GRAVITATE project. As a substitutive solution, a local annotation functionality has been implemented in the desktop client, so that annotations are saved in the user's disk space.

Annotations are coded in the ".jsonld" files and follow the JSON-LD format. JSON-LD is a JSON-derived syntax designed for Semantic Web and Linked Data applications. Besides being a valid JSON document, it also expresses a RDF graph, thanks to the "context" mechanism. This means that a JSON-LD document can be parsed either as JSON or as RDF, according to the needs of the consuming application.

JSON-LD support is already available in triple stores like Blazegraph, and in RDF parsing libraries like RDFLib (Python) and Apache Jena (Java), so it is straightforward to convert a JSON-LD document to another syntax like Turtle, N-Triples or RDF/XML.

Triple stores like Blazegraph, and libraries like RDFLib in Python are able to convert back and forth between JSON-LD and other RDF serializations. Thus, the Blazegraph triple store can be used to store annotations, and make them searchable subsequently. For example, a user which would like to filter all the artefacts containing a head, would filter the part annotations having the "Head" term as "body" of the annotation.

In order to implement the annotation server, as intended in the original GRAVITATE platform, some actions should be taken. First, it is suggested to follow the Web Annotation Protocol[29], which is intended as the recommended client-server communication protocol for an annotation server that handles annotations expressed using the Web Annotation Data Model. For the storage of the annotations, it is recommended to use a triple store, like Blazegraph:using a triple store, the annotations are searchable using the SPARQL language.

The insertion of a new annotation is usually a two-step procedure: the selectors and the corresponding annotations are uploaded separately, because they are in fact two separate resources. To deal with this, the back-end should also offer a file upload service: this way, in the first step, the client uploads the selector file, and receives the URL of the file as response. In the second step, it uploads the annotation containing the URL to the selector resource received in the first step.

Summarizing the components of the annotation server, there should be present:

1. An **API component** that accepts annotations conforming to the Web Annotation Data Model, and sends them to the triple store; it should offer also search facilities;
2. The **triple store**, which stores the annotations;
3. A **file upload** service;
4. The **file repository**, which can be the same repository storing the other 3D files.

[29] https://www.w3.org/TR/annotation-protocol/

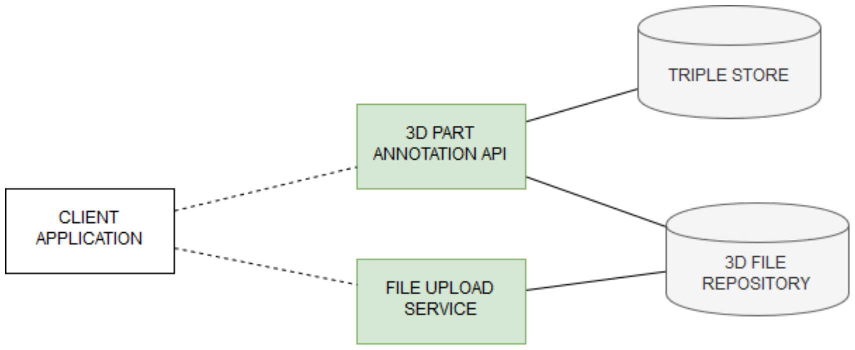


Figure 5: The components of the annotation service

In the current implementation state, the triple store and the 3D file repository are already part of the architecture; the other components (annotation API and file upload service) can be implemented as additional "apps" of the Django server.

This functionality of course affects also the development of the desktop client, as it is required to set up the network requests conforming to the back-end annotation protocol. The expected changes are detailed in sub-section 4.8.

4.2 Annotation format

According to the WebAnnotation data model[30], an annotation has a target (the subject of the annotation) and a body (something "about" the target).

In case of part annotation, the target is a "Specific Resource" object with two properties:

- **source**: the object being annotated. In the case of GRAVITATE, a 3D model, expressed by its URI;
- **selector**: a resource that specifies a *segment of interest* in the **source**; it can be either an embedded resource (as in some WebAnnotation examples), or an external resource[31], denoted by a URI (that is our case).

In the remainder of this document, the words "source" and "selector" are used with this meaning.

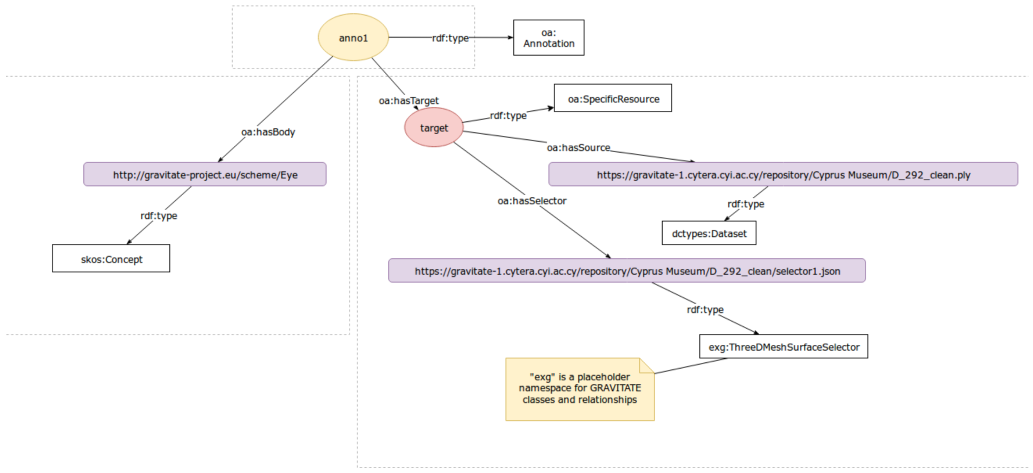

anno1
rdf:type
oa: Annotation
oa:hasTarget
oa:hasBody
target
rdf:type
oa:SpecificResource
oa:hasSource
http://gravitate-project.eu/scheme/Eye
https://gravitate-1.cytera.cyi.ac.cy/repository/Cyprus Museum/D_292_clean.ply
rdf:type
oa:hasSelector
rdf:type
skos:Concept
dctypes:Dataset
https://gravitate-1.cytera.cyi.ac.cy/repository/Cyprus Museum/D_292_clean/selector1.json
rdf:type
exg:ThreeDMeshSurfaceSelector
"exg" is a placeholder namespace for GRAVITATE classes and relationships

Figure 6: A concrete example of part annotation in the GRAVITATE scenario: annotating a part of the "D_292_clean.ply" model with the term "Eye"

Annotations are expressed in the JSON-LD[32] format (see next section for more information)

From https://www.w3.org/TR/json-ld/#introduction:

[30] https://www.w3.org/TR/annotation-model/
[31] https://www.w3.org/TR/annotation-model/#external-web-resources
[32] https://www.w3.org/TR/json-ld/

JSON-LD is a lightweight syntax to serialize Linked Data in JSON [RFC4627]*. Its design allows existing JSON to be interpreted as Linked Data with minimal changes. JSON-LD is primarily intended to be a way to use Linked Data in Web-based programming environments, to build interoperable Web services, and to store Linked Data in JSON-based storage engines*

With JSON-LD, JSON attributes (and values) can be mapped to RDF resources (e.g. classes, properties, individuals, literals). This is done by specifying a so-called "context". Contexts are defined with the special attribute **`@context`**. Contexts can be set inline (as JSON objects) or linked as external resources. For example, the context for the Web Annotation attributes is **`http://www.w3.org/ns/anno.jsonld`**.

4.3 Blazegraph support to JSON-LD

It seems that Blazegraph supports updates in JSON-LD format[33] out-of-the-box. This means that saving a JSON-LD annotation as a set of RDF triples is a straightforward task.

Example:

```
curl -X POST -H 'Content-Type:application/ld+json' --data-binary
'@src/test/java/com/bigdata/rdf/rio/little.jsonld' http://localhost:9999/blazegraph/sparql
```

Security warning: Blazegraph would accept any valid JSON-LD document (even though it does not represent an annotation). This would open doors to a potential abuse of the service. **For this reason, the annotation service should also have a layer of input validation**, to ensure that the JSON-LD document complies with the Web Annotation specification: for instance, checking that the object has "type" attribute equal to "Annotation", and so on.

The type of validation that is required, in order to have a well-formed annotation, is explained here:

https://www.w3.org/TR/annotation-model/#annotations

4.4 Annotating a 3D part

The example described in Figure 6 can be expressed in JSON-LD format:

```
{
    "@context": "http://www.w3.org/ns/anno.jsonld",
    "id": "http://gravitate-1.cytera.cyi.ac.cy/annotations/anno1",
    "type": "Annotation",
    "created": "2015-01-31T12:03:45Z",
    "body": {
      "id": "http://gravitate-project.eu/scheme/Eye",
      "type": "skos:Concept"
    },
    "target": {
          "@context": "http://gravitate-project.eu/scheme/threedanno.jsonld",
          "type": "SpecificResource",
          "source": {
              "id": "https://example.org/D_292_clean.ply",
              "type": "Dataset",
              "format": "model/mesh"
          },
```

[33] https://jira.blazegraph.com/browse/BLZG-1017?focusedCommentId=21127&page=com.atlassian.jira.plugin.system.issuetabpanels:comment-tabpanel#comment-21127

```
          "selector": {
              "id": "https://example.org/selector1.json",
              "type": "ThreeDMeshSurfaceSelector"
          }
    }
  }
```

Links in green denote resolvable URLs:

- **threedanno.jsonld** is the **context** file, which maps the JSON-LD document to a RDF graph and vice versa (example: the WebAnnotation context[34]). The context file should be a resolvable URL; alternately, the context can be embedded in the annotation (useful for early development and tests);
- **D_292_clean.ply** is the 3D model to be annotated;
- **selector1.json** is the **selector file**, which describes the geometry of the part to be annotated.

Note: in the example, the context is linked as an external resource, but it can also be embedded. Example from the JSON-LD specification: https://www.w3.org/TR/json-ld/#embedding.

4.5 Annotation protocol

Although the Web Annotation data model does not prescribe a transport protocol, there is a recommended Web Annotation protocol that can be used. Nevertheless, implementing all the functionalities proposed in the protocol is cumbersome, and out of the scope of the IMATI research objectives. Thus, in this sub-section we propose a minimal implementation of the protocol, which is focused on the part-based annotation of 3D models (i.e. target resources can be only 3D models), and does not consider (but it does not exclude) other media types as valid targets.

The annotation service should be able at least to:

- **Create**, **Update** and **Delete** annotations,
- **Retrieve** available annotations for a given 3D model.

Users of the desktop client see the annotation as an atomic operation; under the hood it is actually composed of multiple steps

1. Create and upload the selector file,
2. Create and upload the annotation.

Annotations are composed of two parts: the actual annotation, and the selector file that is linked as an external resource. For these two parts, there should be two distinct services:

1. A **file upload component** to upload the selector files; selector files should be uploaded before the annotation itself,
2. The **annotation component**, which should be able to **Create**, **Retrieve**, **Update** and **Delete** the annotations.

The **file upload component** accepts files through a POST request, and returns the URL of the file if it has been uploaded correctly.

NOTE: in the current repository structure, annotations should be put in a "Parts/" subdirectory under the 3D model directory (e.g. **/repository/D292_clean/Processed_Models/Parts/**). With a generic file upload service, such position cannot be determined. Therefore, a solution would be to upload the file to a temporary position (returned to the client). The client would then upload the annotation using the temporary position, then the annotation server would take care of moving the file to the new position, and to update the corresponding "selector" attribute. Another

[34] https://www.w3.org/TR/annotation-vocab/#json-ld-context

possible solution is to offer support to annotations with embedded selectors. This way, the annotations and the corresponding selectors are uploaded in a single step; the selector could be extracted from the annotation and saved separately as a file, and linked to the annotation on the server side. Subsequent requests of the annotation will return the linked selector, rather than embedded.

The metadata component is the core of the annotation service. It allows the user to create a new annotation[35], or to update an existing one[36].

4.6 Annotation containers

Annotations are grouped in *Annotation Containers*[37], which are a kind of LDP Containers, with additional constraints. In order to create, update and delete annotations as in a general annotation setting, the protocol requires that the annotation server supports such containers.

In a search scenario, the user should be able to retrieve all the annotations made on a single 3D object. This could be done by setting a query parameter that filters the annotations by target.

Request example (based on the container request example[38]):

```
GET /annotations/?target=https://gravitate-1.cytera.cyi.ac.cy/repository/
Cyprus_Museum/D_292_clean.ply HTTP/1.1
Host: example.org
Accept: application/ld+json; profile="http://www.w3.org/ns/anno.jsonld"
Prefer: return=representation;
include="http://www.w3.org/ns/oa#PreferContainedIRIs"
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/ld+json; profile="http://www.w3.org/ns/anno.jsonld"
Content-Location: http://example.org/annotations/?iris=1
ETag: "_87e52ce123123"
Link: <http://www.w3.org/ns/ldp#BasicContainer>; rel="type"
Link: <http://www.w3.org/TR/annotation-protocol/>;
rel="http://www.w3.org/ns/ldp#constrainedBy"
Allow: POST,GET,OPTIONS,HEAD
Vary: Accept, Prefer
Content-Length: 397

{
  "@context": [
    "http://www.w3.org/ns/anno.jsonld",
    "http://www.w3.org/ns/ldp.jsonld"
  ],
  "id": "http://example.org/annotations/?iris=1",
  "type": ["BasicContainer", "AnnotationCollection"],
  "total": 42023,
```

[35] https://www.w3.org/TR/annotation-protocol/#create-a-new-annotation
[36] https://www.w3.org/TR/annotation-protocol/#update-an-existing-annotation
[37] https://www.w3.org/TR/annotation-protocol/#annotation-containers
[38] https://www.w3.org/TR/annotation-protocol/#representations-with-annotation-descriptions

```
  "modified": "2016-07-20T12:00:00Z",
  "label": "A Container for Web Annotations",
  "first": {
    "id": "http://example.org/annotations/?iris=1&page=0",
    "type": "AnnotationPage",
    "next": "http://example.org/annotations/?iris=1&page=1",
    "items": [
      "http://example.org/annotations/anno1",
      "http://example.org/annotations/anno2",
      "http://example.org/annotations/anno3",
      "http://example.org/annotations/anno4",
      "http://example.org/annotations/anno5",
      "http://example.org/annotations/anno6",
      ...
      "http://example.org/annotations/anno999",
    ]
  },
  "last": "http://example.org/annotations/?iris=1&page=42"
}
```

4.7 CRUD operations on the annotation resources

4.7.1 Creating a new part annotation

From the Web Annotation Protocol:

> ***New Annotations are created via a POST request to an Annotation Container****. The Annotation, serialized as JSON-LD, is sent in the body of the request [...]. The server MAY reject content that is not considered an Annotation according to the Web Annotation specification [annotation-model][39].*
>
> *Upon receipt of an Annotation, the server MAY assign IRIs to any resource or blank node in the Annotation,* ***and MUST assign an IRI to the Annotation resource in the id property, even if it already has one provided****. The server SHOULD use HTTPS IRIs when those resources are able to be retrieved individually.* ***The IRI for the Annotation MUST be the IRI of the Container with an additional component added to the end****.*

See: https://www.w3.org/TR/annotation-protocol/#create-a-new-annotation

Note: With this API, it is also possible to annotate whole models, not just its parts. In order to do this, it is enough to send an annotation where the target **is** the 3D model:

```
{
  "@context": "http://www.w3.org/ns/anno.jsonld",
  "type": "Annotation",
  "body": {
    "id": "http://gravitate-project.eu/scheme/Eye",
    "type": "skos:Concept"
  },
  "target": {
     "id": "https://example.org/D_292_clean.ply",
```

[39] https://www.w3.org/TR/annotation-protocol/#bib-annotation-model

```
      "type": "Dataset",
      "format": "model/mesh"
    }
}
```

4.7.2 Retrieving an annotation

An annotation can be retrieved by sending a GET request to its URI (obtained with a container request).

See: https://www.w3.org/TR/annotation-protocol/#annotation-retrieval

Request example:

```
POST /annotations/ HTTP/1.1
Host: gravitate-1.cytera.cyi.ac.cy
Accept: application/ld+json; profile="http://www.w3.org/ns/anno.jsonld"
Content-Type: application/ld+json; profile="http://www.w3.org/ns/anno.jsonld"
Content-Length: ...

{
  "@context": "http://www.w3.org/ns/anno.jsonld",
  "type": "Annotation",
  "body": {
    "id": "http://gravitate-project.eu/scheme/Eye",
    "type": "skos:Concept"
  },
  "target": {
     "@context": "http://gravitate-project.eu/scheme/threedanno.jsonld",
     "type": "SpecificResource",
     "source": {
         "id": "https://example.org/D_292_clean.ply",
         "type": "Dataset",
         "format": "model/mesh"
     },
     "selector": {
         "id": "https://example.org/selector1.json",
         "type": "ThreeDMeshSurfaceSelector"
     }
  }
}
```

Reply example:

```
HTTP/1.1 201 CREATED
Allow: PUT,GET,OPTIONS,HEAD,DELETE,PATCH
Location: http://gravitate-1.cytera.cyi.ac.cy/annotations/anno1
Content-Type: application/ld+json; profile="http://www.w3.org/ns/anno.jsonld"
Content-Length: ...
ETag: "_87e52ce126126"

{
  "@context": "http://www.w3.org/ns/anno.jsonld",
  "id": "http://gravitate-1.cytera.cyi.ac.cy/annotations/anno1",
  "type": "Annotation",
  "created": "2015-01-31T12:03:45Z",
  "body": {
    "id": "http://gravitate-project.eu/scheme/Eye",
    "type": "skos:Concept"
  },
  "target": {
        "@context": "http://gravitate-project.eu/scheme/threedanno.jsonld",
        "type": "SpecificResource",
        "source": {
            "id": " https://example.org/D_292_clean.ply",
            "type": "Dataset",
            "format": "model/mesh"
        },
        "selector": {
            "id": " https://example.org/selector1.json",
           "type": "ThreeDMeshSurfaceSelector"
        }
  }
}
```

4.7.3 Updating an annotation

From the Web Annotation Protocol:

> *Annotations can be updated by using a PUT request to replace the **entire state** of the Annotation. Annotation Servers SHOULD support this method.*

See: https://www.w3.org/TR/annotation-protocol/#update-an-existing-annotation

Note: Generally, the user would be mostly interested in changing the selector of a part, leaving the annotation untouched. Then, it might be done by just replacing the selector file in the same URL.

However, a conservative approach is recommended, that is, the user uploads a new selector in the repository, and then an annotation update is sent together with the new selector URL.

4.7.4 Deleting an annotation

Deleting an annotation just consists in calling the DELETE method on the resource corresponding to the annotation; no specific comment on it.

See: https://www.w3.org/TR/annotation-protocol/#delete-an-existing-annotation

4.8 Expected changes to the Desktop client

The implementation of a back-end annotation server requires changes also on the client-side. In particular, this is a list of changes that are expected, to have a complete client-server annotation:

- The **`WebAnnotation`** class currently supports only embedded selectors (i.e. the "selector" attribute is JSON object, no other network requests are required); it should be added the support to "linked" selectors (i.e. the "selector" attribute is a URL, which means that the corresponding file should be loaded with a HTTP request).
- A **`BackendAnnotationModel`** class, deriving from **`AnnotationModelBase`**, should be implemented, preferably following the Web Annotation protocol specification.

The retrieval and delete operations are quite straightforward: they are simple calls to the corresponding API endpoints, respectively with the **`GET`** and **`DELETE`** method. Conversely, the creation (**`POST`**) and update (**`PUT`**) operations have some critical point to tackle, which are explained in the remainder of this sub-section.

4.8.1 Creation of a new annotation

The upload of a new annotation may follow two possible alternatives (detailed in sub-section 4.5):

1. Using the two-step procedure (uploading the selector first, then the annotation pointing to the selector file);
2. Uploading an annotation with the embedded selector, implying that the annotation is then converted to the linked form in the back-end.

The trade-off between the first approach and the second, is that the first requires more changes client-side, but less server-side (or at least, a less complicated application logic); conversely, the second approach requires less changes client-side, but a more complex implementation server-side.

In any case, the storage of the annotation should happen in the "linked" form, because the selectors we have defined contain a (potentially) large array of integers. Such array cannot be handled efficiently by a triple store (it would create a triple for each value in the array), and it would be also more difficult to convert back to the JSON-LD format.

4.8.2 Update of an existing annotation (add/remove SKOS terms)

This use case may happen if the user adds or removes a SKOS term from an existing annotation.

Actually, this case is handled very similarly to the creation case, except that a **`PUT`** request is sent, instead of a **`POST`** one. The request body is the new annotation, complete of all the attributes, which replaces totally the old one. If the selector is not changed, it is possible to reuse the same URL, which the client has still in memory, since the existing annotations have been loaded previously.

4.8.3 Update of an existing annotation (change of the part selection)

This case has been added just for completeness, since the desktop client does not have a GUI control to do it. However, this case is much similar to the creation of a new annotation, because the client needs to upload a new selector, which will replace the old one. So, the implementation of this use case is similar to the creation, except for the usage of the **`PUT`** method.

4.8.4 Retrieval of an annotation

As explained before, there are two modalities related to the specification of the selector: the "linked" one (**`selector`** attribute as a URI) and the "embedded" one (**`selector`** attribute as a JSON object). With the proposed implementation of the annotation server, a GET request is expected to return the "linked" form.

5 References

[1] C.E. Catalano, A. Repetto, M. Spagnuolo, M. Bashevoy, D4.1 GRAVITATE Dashboard: Usage Scenarios, (2016). http://gravitate-project.eu/.

[2] A. Repetto, C.E. Catalano, M. Spagnuolo, D4.2 GRAVITATE Dashboard: Mockup, (2016). http://gravitate-project.eu/.

[3] C.E. Catalano, A. Repetto, M. Spagnuolo, A Dashboard for the Analysis of Tangible Heritage Artefacts: a Case Study in Archaeology, in: Eurographics Work. Graph. Cult. Herit., 2017: pp. 151–160. doi:10.2312/gch.20171307.

[4] M. Mortara, C. Pizzi, M. Spagnuolo, Streamlining the Preparation of Scanned 3D Artifacts to Support Digital Analysis and Processing: the GRAVITATE Case Study, in: T. Schreck, T. Weyrich, R. Sablatnig, B. Stular (Eds.), Eurographics Work. Graph. Cult. Herit., The Eurographics Association, 2017. doi:10.2312/gch.20171309.

[5] S. Biasotti, A. Cerri, B. Falcidieno, M. Spagnuolo, 3D artifacts similarity based on the concurrent evaluation of heterogeneous properties, J. Comput. Cult. Herit. 8 (2015) 19.

[6] S. Biasotti, E.M. Thompson, M. Spagnuolo, Experimental Similarity Assessment for a Collection of Fragmented Artifacts, in: Proc. 11th Eurographics Work. 3D Object Retr., Eurographics Association, Goslar Germany, Germany, 2018: pp. 103–110. http://dl.acm.org/citation.cfm?id=3290638.3290655.

[7] S. Biasotti, E.M. Thompson, M. Spagnuolo, Context-adaptive navigation of 3D model collections, Comput. Graph. 79 (2019) 1–13. doi:https://doi.org/10.1016/j.cag.2018.12.004.

[8] A. Repetto, C.E. Catalano, S. Biasotti, E. Moscoso Thompson, M. Spagnuolo, S. Modafferi, M. Polig, D4.3 GRAVITATE Dashboard Components, (2018).

[9] M.-L. Torrente, S. Biasotti, B. Falcidieno, Recognition of feature curves on 3D shapes using an algebraic approach to Hough transforms, Pattern Recognit. (2017). doi:10.1016/j.patcog.2017.08.008.

[10] C.E. Catalano, J. Moffet, D. Oldman, D5.3 The GRAVITATE Data Model, 2018.

[11] R. Sanderson, Web Annotation Protocol, 2017. https://www.w3.org/TR/annotation-protocol/.

[12] S. Modafferi, D5.1 Final Functional Specification and Architecture Design of the GRAVITATE platform, 2016.

[13] M. Attene, F. Giannini, M. Pitikakis, M. Spagnuolo, The VISIONAIR Infrastructure Capabilities to Support Research, Comput. Aided. Des. Appl. 10 (2013) 851–862. doi:10.3722/cadaps.2013.851-862.