



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 870811



Social cohesion, Participation, and Inclusion
through Cultural Engagement

D6.8 APIs Specification and Deployment

Deliverable information	
WP	WP6
Document dissemination level	PU Public
Deliverable type	R Document, report
Lead beneficiary	OU
Contributors	UNIBO, GVAM, PG, UCM, UNITO, CELI, UH, CNR
Date	28/04/2023
Document status	Final
Document version	V1.0

Disclaimer: The communication reflects only the author's view and the Research Executive Agency is not responsible for any use that may be made of the information it contains

INTENTIONALLY BLANK PAGE

Project information

Project start date: 1st May 2020

Project Duration: 36 months

Project website: <https://spice-h2020.eu>

Project contacts

Project Coordinator

Silvio Peroni

ALMA MATER STUDIORUM -
UNIVERSITÀ DI BOLOGNA

Department of Classical
Philology and Italian Studies –
FICLIT

E-mail: silvio.peroni@unibo.it

Project Scientific coordinator

Aldo Gangemi

Institute for Cognitive Sciences
and Technologies of the Italian
National Research Council

E-mail: aldo.gangemi@cnr.it

Project Manager

Adriana Dascultu

ALMA MATER STUDIORUM -
UNIVERSITÀ DI BOLOGNA

Executive Support Services

E-mail:

adriana.dascultu@unibo.it

SPICE consortium

No.	Short name	Institution name	Country
1	UNIBO	ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA	Italy
2	AALTO	AALTO KORKEAKOULUSAATIO SR	Finland
3	DMH	DESIGNMUSEON SAATIO - STIFTELSEN FOR DESIGNMUSEET SR	Finland
4	AAU	AALBORG UNIVERSITET	Denmark
5	OU	THE OPEN UNIVERSITY	United Kingdom
6	IMMA	IRISH MUSEUM OF MODERN ART COMPANY	Ireland
7	GVAM	GVAM GUIAS INTERACTIVAS SL	Spain
8	PG	PADAONE GAMES SL	Spain
9	UCM	UNIVERSIDAD COMPLUTENSE DE MADRID	Spain
10	UNITO	UNIVERSITA DEGLI STUDI DI TORINO	Italy
11	FTM	FONDAZIONE TORINO MUSEI	Italy
12	MAIZE	MAIZE SRL (Previously CELI SRL)	Italy
13	UH	UNIVERSITY OF HAIFA	Israel
14	CNR	CONSIGLIO NAZIONALE DELLE RICERCHE	Italy

Executive summary

SPICE is an EU H-2020 project dedicated to citizen curation of cultural heritage. To support citizen curation, the project research upon and develops an ecosystem of methods and tools co-designed by an interdisciplinary team of researchers, technologists, domain experts, and user communities.

In Work Package 6, we design and implement the formal semantics for an integrated socio-technical system for citizen curation. WP6, jointly with WP4, aims at devising a technical research infrastructure to integrate multiple knowledge graphs and ontologies, a linked data social media layer, interface components, annotation software, recommendation systems, data mining tools, and models/methods devised by the SPICE work packages.

In this deliverable, we report on the final specifications for APIs that have been developed and used in SPICE.

Document History

Version	Release date	Summary of changes	Author(s) -Institution
V0.1	20/03/2023	Document structure preparation, template definition of APIs with instructions for contributors.	Jason Carvalho (OU)
V0.2	06/04/2023	API specifications updated by contributors	All partners
V0.3	11/04/2023	Final contributions made. Report sent for internal review.	All partners
V0.4	21/04/2023	Internal review comments processed and amendments made	Jason Carvalho (OU), Chukwudi Uwasomba (OU), Gautam Vishwanath (DMH)
V0.5	26/04/2023	Final edits and formatting	Jason Carvalho(OU), Enrico Daga (OU)
V1.0	28/04/2023	Final version submitted to REA	UNIBO

Table of Contents

- Project information 3
 - Project contacts..... 3
 - SPICE consortium 3
- Executive summary 4
- Document History 5
- 1 Introduction..... 8
- 2 SPICE Linked Data Hub API 9
 - 2.1 Description of the system 9
 - 2.2 Current applications and pilots 10
 - 2.3 Metadata..... 11
 - 2.4 Guide for developers..... 12
 - 2.4.1 User operations 12
 - 2.4.2 Management operations..... 16
 - 2.5 OpenAPI Specification 17
- 3 User Model API..... 19
 - 3.1 Description of the system 19
 - 3.2 Metadata 19
 - 3.3 Guide for developers..... 19
 - 3.4 Schemas..... 24
- 4 Community Model API 26
 - 4.1 Description of the system 26
 - 4.2 Metadata..... 27
 - 4.3 Guide for developers..... 28
 - 4.3.1 Community operations..... 28
 - 4.3.2 User operations 29
 - 4.3.3 Similarity operations 30
 - 4.3.4 Perspective operations..... 32
 - 4.3.5 VISIR operations 33
 - 4.3.6 Development operations..... 34
 - 4.4 OpenAPI Specification 36
- 5 SPICE Semantic Annotator API 37
 - 5.1 Description of the system 37
 - 5.2 Metadata 39
 - 5.3 Guide for developers..... 39
- 6 Social Recommender API 43
 - 6.1 Description of the system 43

6.2	Metadata.....	43
6.3	Guide for developers.....	43
6.4	API Schemas	45
7	Ontology server, query and reasoning services	46
7.1	Ontology Uploading and Reasoning calls via OWL-API (Steps 1-2).....	47
7.2	Ontology Export as JENA Triple-based graph model (step 3)	48
7.3	External Exposure of the Graph Model in a SPARQL Server (step 4)	48
7.4	How to QUERY and UPDATE the exposed Fuseki 2 Model with SOH	49
7.5	How to QUERY and UPDATE the exposed Fuseki 2 Model with RDF Connection	50
7.6	An overview of the API and workflow architecture for DEGARI GAMStories.....	52
7.7	The workflow architecture for DEGARI GAMStories.....	53
7.8	Insert and update Fuseki with GAM stories	54
7.9	Dataset stored in the Linked Data Hub (LDH)	56
7.10	Graphical Interface	56
7.11	Source code	58
8	Conclusions.....	59

1 Introduction

SPICE is an EU H-2020 project dedicated to citizen curation of cultural heritage. To support citizen curation, the project research upon and develops an ecosystem of methods and tools co-designed by an interdisciplinary team of researchers, technologists, domain experts, and user communities.

In Work Package 6, we design and implement the formal semantics for an integrated socio-technical system for citizen curation. WP6, jointly with WP4, aims at devising a technical research infrastructure to integrate multiple knowledge graphs and ontologies, a linked data layer, interface components, annotation software, recommendation systems, data mining tools, and models/methods devised by the SPICE work packages.

In this deliverable, we describe the APIs that have been developed and released for use within the SPICE project architecture. These APIs are predominantly used by pilot applications to provide access to data storage and retrieval, linked data operations, ontologies and reasoning services. The report provides the final API specifications along with software repository links where available. This, together with D4.7 which describes the final version and deployment instructions for the SPICE Linked Data Hub, gives a complete overview of the structure, installation and use of the technologies and software platforms used within the SPICE project.

The deliverable is structured as follows. For each API, this document introduces an overview of its intended purpose in relation to SPICE work packages and any relevant design methodology employed. This includes a report on how each API is currently being used within the SPICE project and, specifically, which pilot applications are making use of its features. Details of the technical underpinnings of the APIs are provided. Further background information on each API can be found in the dedicated deliverables, these are referenced in their corresponding sections. For each section, a guide for developers is made available that details the functions available within each API and how to use and configure them through the use of appropriate parameters. Example API requests and code snippets are provided. The APIs detailed in this report are the **SPICE Linked Data Hub API**, the **User Model API**, the **Community Model API**, the **SPICE Semantic Annotator API** and the **Social Recommender API**. While these components refer to systems already described in other deliverables, in Chapter 7 we present the **Ontology Server** as both system specification and running instance. As such, the chapter has a more detailed structure. Section 8 summarises and concludes the report.

The contents of this report were presented in an interim state at the end of the second year of the project, in deliverable D6.4. The contents of this deliverable therefore extend D6.4; this deliverable being the final version of the specifications that were laid out a year ago, incorporating any developments to these technologies that have taken place since then.

2 SPICE Linked Data Hub API

2.1 Description of the system

The SPICE Linked Data Hub (LDH) was developed as a data infrastructure to support the acquisition and management of dynamic data from a variety of sources including: museum collection metadata and digital assets, social media events and user activities, systems' activities (e.g., recommendations, reasoning outputs), ontologies and linked data produced by pilot case studies.

The SPICE Linked Data Hub API is an instance of the API Factory software and underpins the functionality of the LDH Portal; the front-end web-interface component of the LDH. The LDH-API exposes a selection of REST-based user operations for creating, managing and consuming data, as well as management functions which are used by the web portal for creating datasets and managing permissions. The API's position within the wider LDH system is shown in Figure 2.1.1

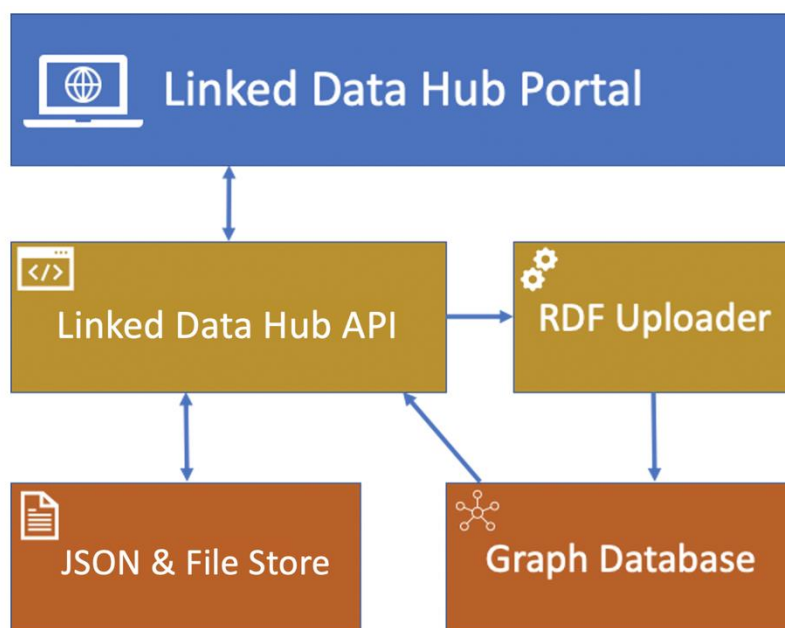


Figure 2.1.1: SPICE Linked Data Hub layout

As well as forming the technical backbone of the LDH portal, the LDH-API offers direct access to data storage and retrieval functions for SPICE application and pilot developers, through a series of REST endpoints. Full documentation on these API functions with example requests are included below in the *Guide for developers*.

The LDH-API operates primarily with JSON documents and makes use of MongoDB as its main data store. The LDH-API also supports storage and retrieval of files (binary and text-based) and associated metadata. Single datasets within the LDH can host a mixture of both JSON documents and binary files.

Parallel to the JSON data store is a graph database. All data within the LDH-API is also replicated into this graph database as RDF and made available through the API via read-only SPARQL queries.

Since deliverable D6.4, the main development to the LDH API has been the addition of the *changes* endpoint. As described in section 2.4, this enables dataset users to monitor when changes happen to datasets. This has been used by third party applications as a trigger to perform external processing operations on the updated data such as running data reasoning services or other change-driven processes such as replication and conversion to external linked data stores. Since the LDH API was required to be in place for the development of SPICE pilot applications, the bulk of the development was done in the early stages of the project. Year

three development of the LDH infrastructure has largely taken place on the LDH portal, described in detail in D4.7.

2.2 Current applications and pilots

There are currently around 30 SPICE datasets accessible via the API that have been collected in the context of the following pilots:

- **IMMA Viewpoints**
 IMMA Viewpoints is a mobile web application that encourages visitors to share their own response to artworks within the grounds of the Irish Museum of Modern Art (IMMA). The IMMA Viewpoints web application makes use of the LDH-API for file storage of artwork images and uses the JSON store for application configuration and the collection, moderation and review of user responses.
- **IMMA Deep Viewpoints**
 Deep Viewpoints builds on the original IMMA Viewpoints prototype, making use of the IMMA Collection dataset stored on the Linked Data Hub. With Deep Viewpoints users can add artworks from the IMMA collection to their own personal collections. Deep Viewpoints has a separate dataset for storing data added by the users. This includes:

 - personal artwork collections
 - scripts and themes
 - users and passwords
 - data generated when a user undertakes a script (e.g. answers to questions)
 - moderation status of contributions
- **InSpice**
 A web framework for the creation of citizen curation activities within the different museums involved in SPICE. It proposes a template-based use model whereby museum curators can instantiate, manage and publish activities based on one of the templates provided by the framework. In this context, the LDH is used primarily as a storage, query and management space for the JSON files used to define specific instances of framework activities, as well as to access the various works and artefacts of the museums involved that already have an associated collection within the LDH.
- **Hecht Museum – Student Experience**
 School students before, during and after a museum school trip at the Hecht Museum learn about their country's history and at the same time learn about the diversity of opinions regarding historical and national issues. Students learn to interpret museum artefacts according to their own personal views, reflect upon other students' opinions, connect their opinions with tangible artefacts at the museum, and perform citizen curation activities.
 The LDH infrastructure is used within this pilot for managing 5 major objects with basic CRUD (Create, Read, Update, Delete) operation: Users, User History, User Model Properties, User Generated Content and Sourced Content.
- **Design Museum Helsinki – Pop-up VR Museum**
 The focus of the Design Museum Helsinki (DMH) Case Study is on developing the citizen curation methods by first gathering interpretations of DMH collection objects in workshops with selected end-user communities, namely senior citizens, remote dwellers, and asylum seekers (D7.3, pg33). An

application known as the *Pop-up VR Museum* has been designed and is accessible to audiences via portable VR headsets. Its users can access, interact, and engage with Design Museum Helsinki’s collections.

The experience of the Pop-up VR Museum is bound to be generative and dictated by a dynamic online repository of artefacts (3D models) and narratives (audio recordings and textual data) stored in the LDH. Mediators such as DMH staff, researchers, and members of affiliated institutions add artefact ontologies and narratives collected from end-user contributors to the LDH.

- **GAM Game**

The case study of the Gallery of Modern Art (GAM) in Turin, which addresses the inclusion of deaf people as target community, revolves around the notion of storytelling. Through the web app, called GAM Game, users can create short stories by collecting and sequencing the artworks from the museum collection, and add a personal response to each of the artworks in the story. The LDH infrastructure is used to manage three main entities in the interaction with the client:

- User id and data, which include the links to the stories created by the user;
- Stories, each including its own properties (date, title, etc.) and the list of links to artworks by which it is composed, and the user responses (stored in textual form) associated to each artwork in the story;
- Artefacts, each accompanied by its metadata, which include both the ones extracted from the collection catalogues and the ones added by the sensemaking components (associated emotions, values, themes, etc.)

- **Madrid - Treasure Hunt**

The case study of the Natural Museum of Natural Sciences of Madrid (MNCN) revolves around treasure hunts in the museum. A treasure hunt consists of a series of searches guided by clues describing the object in the collection to be found. Once the object is found, the game provides relevant information related to it and may pose related questions.

The LDH infrastructure is used in this pilot to manage a series of entities in its interaction with the client, such as user information, treasure hunt definitions, persistent application state information, artefacts referenced in treasure hunts and user responses and interactions.

2.3 Metadata

id	LDH-API
name	The Linked Data Hub API
description	The data API underpinning the SPICE Linked Data Hub. Supports reading writing and querying of JSON documents and binary files and querying of RDF data representations in SPARQL.
type	API
release-date	27/03/2023
release-number	v0.9.5 (main API) v0.1.6 (SPARQL addon)
work-package	WP4, WP6

pilot	ALL
keywords	JSON, RDF, SPARQL, REST
licence	Apache Licence 2.0
release-link	https://github.com/mkdf/api-factory/releases/tag/v0.9.5 https://github.com/mkdf/api-factory-sparql/releases/tag/v0.1.6
demo	https://api2.mksmart.org/ (access key required)
links	https://github.com/mkdf/api-factory/ (main code repository) https://github.com/mkdf/api-factory-sparql (SPARQL addon)
running-instance	http://spice.kmi.open.ac.uk
credits	Jason Carvalho (jason.carvalho@open.ac.uk) Enrico Daga (enrico.daga@open.ac.uk)

2.4 Guide for developers

The following is an overview of API functions. Full details on specific parameter usage are available via the LDH-API's live interface at <https://api2.mksmart.org/>. Full OpenAPI specification is also referenced in the OpenAPI specification section below.

Access to and usage of the LDH-API is via API keys. API keys can be registered on the LDH web portal and assigned to specific datasets for either read, write or read/write access depending on the access control limitations set by the dataset owner. All API calls must be authenticated, using HTTP Basic Authentication, using the API key as both the username **and** password.

The API is made up of both user and management operations. Management operations are reserved for the creation and management of datasets, user keys and permissions. These management features are only available to API administrators and are also used as an interface to the API by the LDH web portal.

2.4.1 User operations

BROWSE

A read-only API endpoint for retrieving data. The endpoint provides options for paging, sorting, filtering, field selection and complex database queries using MongoDB-style JSON queries.

Usage:

GET `/browse/{dataset-uuid}`

Parameters:

- **query** The filter query
- **sort** Specify fields on which to sort the data. Sort fields should be specified as a comma separated list. Data will be sorted in ascending order. To specify a field to sort in descending order precede that field with a minus ('-')
- **fields** Specify which fields to return. Fields should be specified as a comma separated list. Fields preceded with a minus ("-") will be excluded from the results. The "_id" field is always returned, unless explicitly excluded.
- **pagesize** Specify page size (defaults to a page size of 100)
- **page** Specify the page number of results to return (defaults to page 1)

Example request

```
curl -X GET
"https://api2.mksmart.org/browse/123456789?sort=id&fields=id,value&pagesize=5&page=1" -H "accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0"
```

OBJECT

The object endpoint is used for standard CRUD-style database operations; reading, writing, updating and deleting. The HTTP method used (GET/POST/UPDATE/DELETE) defines which function is called.

Usage:

GET /object/{dataset-uuid} Retrieve documents from the dataset

Parameters

- **query** The filter query
- **limit** Limit the number of documents retruend (defaults to 100)

POST/object/{dataset-uuid} Create a new document in the dataset

GET /object/{dataset-uuid}/{doc-id} Retrieve a single document from the dataset

PUT /object/{dataset-uuid}/{doc-id} Update a document by ID

DELETE /object/{dataset-uuid}/{doc-id} Delete a document by ID

Example requests

Retrieve the most recent 5 objects from dataset 123456789:

```
curl -X GET "https://api2.mksmart.org/object/123456789?limit=5" -H "accept: */*"
-H "Authorization: Basic dGVzdDp0ZXN0"
```

Post a new object to dataset 123456789:

```
curl -X POST "https://api2.mksmart.org/object/123456789" -H "accept: */*" -H
"Authorization: Basic dGVzdDp0ZXN0" -H "Content-Type: application/json" -d
"{\"_id\": \"1067\", \"attribute1\": \"42-a\", \"attribute2\": 34.7}"
```

Retrieve object 1001 from dataset 123456789:

```
curl -X GET "https://api2.mksmart.org/object/123456789/1001" -H "accept: */*" -H
"Authorization: Basic dGVzdDp0ZXN0"
```

Update object 1001 in dataset 123456789:

```
curl -X PUT "https://api2.mksmart.org/object/123456789/1001" -H "accept: */*" -H
"Authorization: Basic dGVzdDp0ZXN0" -H "Content-Type: application/json" -d
"{\"_id\": \"1067\", \"attribute1\": \"42-a\", \"attribute2\": 34.7}"
```

Delete object 1001 from dataset 123456789:

```
curl -X DELETE "https://api2.mksmart.org/object/123456789/1001" -H "accept: */*"
-H "Authorization: Basic dGVzdDp0ZXN0"
```

FILE

This API endpoint is used to manage binary files within a dataset.

Usage:

GET `/file/{dataset-uuid}` Retrieve a list of files for a single dataset

POST`/file/{filename}` Upload a new file

Parameters (supplied as multipart/form-data)

- `title` File title
- `description` File description
- `file` The binary file

GET `/file/{filename}/{doc-id}` Retrieve a single file

POST`/file/{filename}/{doc-id}` Update an existing file

Parameters (supplied as multipart/form-data)

- `title` File title
- `description` File description
- `file` The binary file

DELETE `/file/{filename}/{doc-id}` Delete a file

Example requests

Retrieve a list of files for dataset 123456789

```
curl -X GET "https://api2.mksmart.org/file/123456789" -H "accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0"
```

Upload a new file to dataset 123456789

```
curl -X POST "https://api2.mksmart.org/file/123456789" -H "accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0" -H "Content-Type: multipart/form-data" -F "title=Image Title" -F "description=Image Description" -F "file=@myImage.jpg;type=image/jpeg"
```

Retrieve myImage.jpg from dataset 123456789

```
curl -X GET "https://api2.mksmart.org/file/123456789/myImage.jpg" -H "accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0"
```

Update myImage.jpg in dataset 123456789

```
curl -X POST "https://api2.mksmart.org/file/123456789/myImage.jpg" -H "accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0" -H "Content-Type: multipart/form-data" -F "title=New Title" -F "description=New Description" -F "file=@myImage.jpg;type=image/jpeg"
```

Delete image myImage.jpg from dataset 123456789

```
curl -X DELETE "https://api2.mksmart.org/file/123456789/myImage.jpg" -H "accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0"
```

CHANGES

The changes endpoint of the API can be used to query the Linked Data Hub's activity log for updates to a single dataset. Create, update and delete operations are returned. By default, the newest items are returned first.

Usage:

GET `/changes/{dataset-uuid}` Retrieve a list of changes that have been made to a single dataset.

Optional parameters (supplied as URL query parameters)

- `document-id` Only return changes to a single document
- `timestamp` Only return entries that have occurred since this timestamp
- `limit` The maximum number of entries to return
- `sort` Setting this parameter to '1' reverses the sort order of results to return the oldest items first

Example requests

Retrieve a list of changes for dataset 1234567

```
curl -X GET "https://api2.mksmart.org/changes/1234567" -H "accept: */*" -H "Authorization: Basic ZHNmc2RmOnNkZnNkZg=="
```

Retrieve a list of changes for document 'doc05678', within dataset 1234567, since 01/01/2023 and ordered oldest to newest

```
curl -X GET "https://api2.mksmart.org/changes/1234567?document-id=doc-56789&timestamp=1672531200&sort=1" -H "accept: */*" -H "Authorization: Basic ZHNmc2RmOnNkZnNkZg=="
```

SPARQL

The SPICE Linked Data Hub API primarily operates with data in JSON format. However, all data that is pushed into the LDH is also replicated to RDF graphs so that it can be queried as linked data, using SPARQL. This API endpoint provides the facility to use read-only SPARQL queries against data stored within the LDH.

Usage:

GET `/query/{dataset-uuid}/sparql`

Parameters

- `query` The SPARQL query string (URL encoded)

Example request

This SPARQL query ...

```
SELECT * WHERE { ?s ?p ?o } LIMIT 5
```

... would be made on dataset 123456789 using the following HTTP request:

```
curl -X GET
```

```
"https://api2.mksmart.org/query/123456789/sparql?query=SELECT%20%2A%20WHERE%20%7B%20%3Fs%20%3Fp%20%3Fo%20%7D%20LIMIT%205" -H "accept: application/sparql-results+json" -H "Authorization: Basic dGVzdDp0ZXN0"
```

The results format can be chosen by passing an **Accept** header with the HTTP request. The following header values are supported:

- application/sparql-results+json
- application/sparql-results+xml
- text/csv
- text/tab-separated-values

2.4.2 Management operations

DATASETS

Used for managing datasets within the API.

Usage:

GET `/management/datasets` Retrieve a list of all datasets

POST `/management/datasets` Create a new dataset

Parameters (supplied in the request body):

- **dataset-uuid** The ID of the new dataset
- **key** The initial key to assign for use with this dataset which will be given read/write access. The key will be created if it does not already exist.

GET `/management/datasets/{dataset-uuid}` Retrieves a single dataset summary, including the number of documents in that dataset

Example requests

Retrieve a list of datasets

```
curl -X GET "https://api2.mksmart.org/management/datasets" -H "accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0"
```

Create a dataset with id 123456789 using key *key-001*

```
curl -X POST "https://api2.mksmart.org/management/datasets" -H "accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0" -H "Content-Type: application/x-www-form-urlencoded" -d "dataset-uuid=123456789&key=key-001"
```

Retrieve details for dataset 123456789

```
curl -X GET "https://api2.mksmart.org/management/datasets/123456789" -H "accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0"
```

PERMISSIONS

Used for creating, assigning and managing key permissions on datasets

Usage:

GET /management/permissions Retrieve all permissions

GET /management/permissions/{key} Retrieve all permissions for a single key

POST /management/permissions/{key} Set/update permissions. If the key specified does not already exist, it will be created

Parameters (supplied in the request body):

- **dataset-id** The ID of the new dataset to set permissions on
- **read** Whether this key should have read permission – set to 0 or 1
- **write** Whether this key should have write permission – set to 0 or 1

Example requests

Assign key *key-001* read and write permissions to dataset 123456789

```
curl -X POST "https://api2.mksmart.org/management/permissions/key-001" -H
"accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0" -H "Content-Type:
application/x-www-form-urlencoded" -d "dataset-id=123456789&read=1&write=1"
```

Assign key *key-001* read-only permission to dataset 123456789

```
curl -X POST "https://api2.mksmart.org/management/permissions/key-001" -H
"accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0" -H "Content-Type:
application/x-www-form-urlencoded" -d "dataset-id=123456789&read=1&write=0"
```

Remove all permissions for key *key-001* on dataset 123456789

```
curl -X POST "https://api2.mksmart.org/management/permissions/key-001" -H
"accept: */*" -H "Authorization: Basic dGVzdDp0ZXN0" -H "Content-Type:
application/x-www-form-urlencoded" -d "dataset-id=123456789&read=0&write=0"
```

2.5 OpenAPI Specification

The LDH-API has been built with an OpenAPI specification. By doing so, a number of tools become available that can automatically interpret the specification and generate resources to speed up the use and adoption of the API by application developers. These resources include automatically generated web interfaces and automatically generated API clients in a number of programming languages.

The LDH-API uses a dynamically generated OpenAPI specification that is dependent on any add-on modules that are currently being used. In addition to the core API software, the SPICE LDH-API also makes use of the optional SPARQL add-on, as detailed in the API metadata table above.

A static version of the full OpenAPI 3.0.1 specifications in JSON format has been materialised for the purposes of this report, based on v0.9.5 of the main API software and version v0.1.6 of the SPARQL add-on. These are available within the main API repository for the user operations and management operations in the following file locations:

/module/apif-core/view/apif/core/index/swagger-config-main.json

/module/apif-core/view/apif/core/index/swagger-config-management.json

The permanent link for the repository as of the release of this report is available here:

<https://doi.org/10.5281/zenodo.7858954>

3 User Model API

3.1 Description of the system

The purpose of the User Model (UM) is to store information about the user so that it can be reasoned about in a uniform way, for use in the community model and the case study application. It can also be used to guide scripts and post-analysis.

The developer is exposed to a number of data objects (Users, User Properties, User Generated Content, User History) which can be accessed by basic CRUD (Create, Read, Update, Delete) functionality. The API provides examples of how to derive User Model properties from user interactions and user-generated content.

For information about design principles see D3.1 and D3.3 documents.

A typical scenario could be the following: content is shown to the user, s/he then writes something, this is analysed by the Semantic Analyzer (cf. D3.2), the values derived are stored in the User Model for further use by the Community Model. To show the next bit of content the Social Recommender is used, (whose actions are based on the Community & User Models)

Within the Hecht Case Study, the User Model is used for two applications: 1) A student application which guides them through activities prior to the visit, during the Museum visit and after the museum visit. 2) A teacher/researcher application that allows the teacher/researcher to examine the results and perform some basic analysis. The User Model is also currently used across a number of other case studies, including the work with DMH.

For more details see deliverable D3.3.

3.2 Metadata

Id	UM-API
Name	User Model API
Type	API
release-date	01/05/22
release-number	2.0
work-package	WP3
keywords	User Model, User Profile, REST
Licence	Apache Licence 2.0
demo	User Model Demo, Hecht Use Case
running-instance	
credits	Alan J. Wecker (ajwecker@gmail.com) Tsvika Kuflik (tsvikak@is.haifa.ac.il)

3.3 Guide for developers

In general, the flow is that the user interface (UI) stores a number of User History (UH) items, these can be used to create User Generated Content (UGC). These are analyzed by the Semantic Analyzer (SA) to create User Model Properties (UM). Alternatively, UM properties can be created directly from UH items.

See D3.3 for further detailed information.

user-controller

Usage:

POST /api/v2/users2Create - Create a new user. Note: ID should be anonymized

Parameters:

- No parameters

Request body:

- **User** – see Schema (note: **propertiesCount**, **ugcCount**, **userHistoryCount** are not needed as they are derived values)

Returns:

- 200 – OK

GET/api/v2/users2 - Get all users, sorted by name

Parameters:

- No parameters

Request Body:

- None

Returns:

- 200 – OK. Returns an array of Users

DELETE/api/v2/users2Delete/{userid} - Remove a user by userid

Parameters:

- **Userid**

u-history-controller

Usage:

PUT /api/v2/uhistoryUpdate/{userid} - Update a specific uhistory for a specific user

Parameters:

- **Userid**

Request Body:

- **UHistory**

POST /api/v2/uhistoryCreate/{userid} - Add a new uhistory for a user

Parameters:

- **Userid**

Request Body:

- **UHistory**

Returns:

- None

GET `/api/v2/uhistoryGetAllByUserid/{userid}` - Get all properties for a specific user

Parameters:

- `Userid`

Returns:

- `UHistory`

GET `/api/v2/uhistoryGetAllByPname/{pname}` - Get all properties with a certain uhistory name

Parameters:

- `Pname` - Property name

Request Body:

- None

Returns:

- `UHistory`

GET `/api/v2/uhistoryGet/{userid}/{pname}` - Get a uhistory for a user with a specific name

Parameters:

- `Userid`
- `Pname`

Request Body:

- None

Returns:

- Array of `UHistory`

DELETE `/api/v2/uhistoryDelete/{userid}/{pname}` - Delete a specific uhistory for a specific user

Parameters:

- `Userid`
- `Pname` – property name

Request Body:

- None

Returns:

- Array of `UHistory`

property-controller

Usage:

POST `/api/v2/propertyCreate/{userid}` - Add a new property for a user

Parameters:

- `Userid`: string

- **Pname**: string – property name

Request Body:

- Schema of Property

Returns:

- None

GET `/api/v2/propertyModel/{userid}` - Model properties for a specific user

Parameters:

- **Userid**: string

Request Body:

- None

Returns:

- None

GET `/api/v2/propertyGetAllByUserid/{userid}` - Get all properties for a specific user

Parameters:

- **Userid**

Request Body:

- None

Returns:

- Array of **Property**

GET `/api/v2/propertyGetAllByPname/{pname}` - Get all properties with a certain property name

Parameters:

- **Pname** – property name

Request Body:

- None

Returns:

- Array of **Property**

GET `/api/v2/propertyGet/{userid}/{pname}` - Get a specific property for a specific user

Parameters:

- **Userid**
- **Pname** – property name

Request Body:

- None

Returns:

- `Property`

DELETE `/api/v2/propertyDelete/{userid}/{pname}` - Delete a property for a specific user

Parameters:

- `Userid`
- `Pname` – property name

Request Body:

- None

Returns:

- None

user-generated-content-controller

Usage:

POST `/api/v2/ugcCreate/{userid}` - Add a new User Generated Content for a user

Parameters:

- `Userid`

Request Body:

- `UGC`

Returns:

- None

GET `/api/v2/ugcAnalyzeAllByUser/{userid}` - Analyse all UGCs for a specific user

Parameters:

- `Userid`: string

Request Body:

- None

Returns:

- None

GET `/api/v2/ugcGetByUserIdAndName/{userid}/{ugcname}` - Get UGC for a specific user

Parameters:

- `Userid`
- `Ugcname` – ugc type name

Request Body:

- None

Returns:

- **UGC**

GET /api/v2/ugcGetAllByUserid/{userid} - Get all UGC for a specific user

Parameters:

- **Userid**

Request Body:

- None

Returns:

- Array of **UGC**

3.4 Schemas

The following schema definitions describe the properties names and types for each of the data structures used within the User Model API.

User

source	String – Where was this info taken
context	String – Under what circumstances
id	integer(\$int64)
userid	string
password	string
ptype	String – participation type
role	String - (e.g. visitor, curator)
propertiesCount	integer(\$int64) - Derived number of user model properties
ugcCount	integer(\$int64) - Derived number of user generated content
getuHistoryCount	integer(\$int64) - Derived number of user history items

UHistory

source	string
context	string
id	integer(\$int64)
userid	string
pname	String- Property name
pvalue	String – Property Value

Property

source	String: From which ugc derived
context	application
id	integer(\$int64)
userid	String: from user db
category	String: part of property name (generic)
pname	String: property name from ontology
pvalue	String: property value from ontology
datapoints	integer(\$int32)
origin	String: what artefact is this talking about

UserGeneratedContent

source	string
context	string
id	integer(\$int64)
userid	string
parentname	String- name of parent ugc if this is a comment to comment
parenttype	String – what type of media was parent
contentType	String – what is media type of this entry
ugcname	String – Generic Name of entry
ugcimage	String – If UGC contains image hled here
ugctags	String – User provided tags concerning ugc
ugcdesc	String – Written description of UGC passed to SA
ugcmeta	String – Explicit structured info by user about ugc
emotions	Array of name-value pairs
sentiments	Array of name-value pairs

4 Community Model API

4.1 Description of the system

The Community Model API (CM-API) is the access point to the Community Model (CM), which supports the social cohesion across groups, by the understanding of their differences and recognizing what they have in common. The community model is responsible for storing information about explicit communities that users belong to. Additionally, it creates the implicit communities inferred from user interactions and it computes the metrics needed to define the similarity (and dissimilarity) among group of users. The Community Model will support the recommender system in the variety and serendipity to the recommendation results, that will not be oriented to the typically popular contents or based on providing “more of the same” similar contents to the users (the so called, filter bubble) but to the inter-group similarities and the intra-group differences.

The CM-API exposes a set of REST-based operations for accessing information about implicit and explicit communities, for creating and accessing perspectives and endpoints for operations related to similar and dissimilar communities. CM-API is also employed by the User Model to notify changes in user attributes and the creation of new user generated content. This new version also includes new endpoints for VISIR (a visualization tool for supporting the Interaction-Reflection Loop and the creation of perspectives for the Community Model), as long as some endpoints for development purposes. The CM-API acts as a façade that hides the modules that appear in Figure 4.1.1

The CM-API primarily works using the community data stored in a Document Database implemented using MongoDB¹. The access to this database is implemented using a Data Access Object (DAO) pattern², so it isolates the details for accessing the database. The information stored in the database is generated by the Community Model, described in Deliverables D3.3 and D3.5.

The CM-API is also responsible for supplying the Community Model with all the data about the users within the system. When a new user arrives at the SPICE infrastructure, the User Model notifies and provides the CM-API with the data provided by the user. Additionally, when a user contributes with interactions that are interesting according to the design of the SPICE case studies (interactions with museum items and concepts), the User Model will send these interactions to the Community Model using the CM-API. The Community Model is responsible for managing this new user data, updating the communities for the case study and store them in the database. This way, updated information about the communities will be accessible through the CM-API.

The CM-API will support the recommender system for accessing similarities and differences both within single groups and between different groups. CM-API will not be only used to know which communities a user belongs to, but also to retrieve de most similar o dissimilar communities to a given one. This information is not computed by the CM-API but created and stored by the community model and retrieved from the database.

¹ <https://www.mongodb.com/>

² <https://www.oracle.com/java/technologies/dataaccessobject.html>

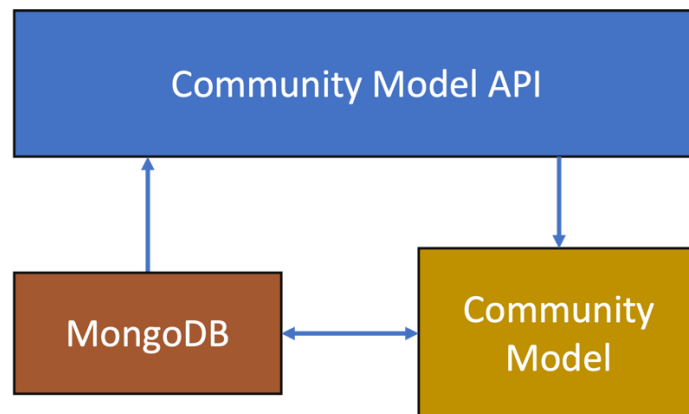


Figure 4.1.1. Overview of the CM-API infrastructure.

4.2 Metadata

id	CM-API
name	Community Model API
description	API for accessing data generated by the Community Model and for updating user data used by the Community Model for finding communities
type	API
release-date	01/03/2022
release-number	v.2.0
work-package	WP3
keywords	Community detection, clustering, REST.
licence	Apache Licence 2.0
links	https://spice.fdi.ucm.es/hecht/ (Documentation) https://spice.fdi.ucm.es/hecht/v2.0/ (API) https://spice.fdi.ucm.es/hecht/api-docs/ (interactive documentation)
running-instance	https://spice.fdi.ucm.es/hecht/ https://spice.fdi.ucm.es/gam/ https://spice.fdi.ucm.es/dmh/ https://spice.fdi.ucm.es/mncn/
credits	Guillermo Jiménez Díaz (gjimenez@ucm.es), Belén Díaz Agudo (belend@ucm.es) José Ángel Sánchez Martín (josanc16@ucm.es), Ilya Lapshin (ilapshin@ucm.es)
related-components	UM-API

4.3 Guide for developers

The following is an overview of API functions. Full OpenAPI specification is also referenced in the OpenAPI specification section below.

The source code repository contains a suite of integration tests that verifies that all the API functions are working correctly according to its OpenAPI specification.

The API provides several entry points with their corresponding operations. Some API requests take some time because the community model needs to update the data and models for providing a response. In this case, the API will return a 202-response code with a job id. With this job ID, the user can periodically monitor the status of the job (using the `GET /jobs-manager/jobs/{id}` endpoint) and retrieve the requested data when the community model finishes the computation process.

All API calls must be authenticated, using HTTP Basic Authentication, using username **and** password. Authentication credentials are created specifically for each case study and, hence, community model server.

4.3.1 Community operations

A read-only API endpoint for retrieving data about the communities.

Usage:

`GET /communities`

Access to a list of all the communities in the community model

Parameters

None

Responses

A list of the communities contained in the community model. Every community is represented in an object with the following attributes:

- **id**: Unique id for the community in the community model
- **name**: Community name (for explicit communities)
- **explanation**: Community description (maybe empty). It can be computed by the explanation module from the Community Model or it can be provided by curators when defining explicit communities.
- **community-type**: Type of community (implicit or explicit). Implicit communities are computed by the community model. Explicit communities are provided by the user model.
- **users**: A list with the user ids who belong to the community.

Example request

```
curl -X GET "https://spice.fdi.ucm.es/hecht/v2.0/communities"
```

`GET /communities/{community-id}`

Returns information about a community

Parameters

None

Responses

An object with information about the requested community.

A response with status 400 is generated if the community model does not store any community with this community id.

Example request

```
curl -X GET
"https://spice.fdi.ucm.es/hecht/v2.0/communities/621e53cf0aa6aa7517c2afdd"
```

`GET /communities/{community-id}/users`

Returns a list with the ids of the users who belong to a community.

Parameters

None

Responses

A list of user ids (cannot be empty).

A response with status 400 is generated if the community model does not contain any community with this community id.

Example request

```
curl -X GET
https://spice.fdi.ucm.es/hecht/v2.0/communities/621e53cf0aa6aa7517c2afdd/users
```

4.3.2 User operations

This endpoint provides GET/POST operations for requesting the communities that a user belongs to and updating data about user attributes and user generated content.

`GET /users/{user-id}/communities`

Returns a list with the communities that the user belongs to.

Parameters

None

Responses

A list of objects with information about the communities that the user belongs to (cannot be empty).

A response with status 400 is generated for invalid user ids.

Example request

```
curl -X GET "https://spice.fdi.ucm.es/hecht/v2.0/users/23/communities"
```

`POST /users/{user-id}/update-generated-content`

This service is employed to inform the Community Model about the User Generated Content (UGC) updated in the User Model.

Request Body

A list of UGC objects that represent the information about the user that will be added to the community model. Every UGC follows the schema provided by the User Model (User Generated Content Schema in section **Error! Reference source not found.**) is represented in an object with the following attributes:

- `id`: Unique id for the UGC in the user model
- `userid`: Unique user id
- `origin`: Unique id for the item or concept in the museum that this user generated content refers to.

- **source_id**: Unique id for the UGC that this content is derived from.
- **source**: Description about the UGC that this content is derived from.
- **pname**: Name of the property included in this UGC.
- **pvalue**: Value of the property included in this UGC.
- **context**: Context of the property included in this UGC.
- **datapoints**: Number of datapoint used to generate this UGC

Only (id, userid, origin, source_id, pname, pvalue) are mandatory required.

Responses

A response with status 400 is generated if the user-id in the url differs from any userid in the objects contained in the list in the body request.

If correct, an empty response with status 204 is generated.

Example request

```
curl -d
' [{"id": "12345", "userid": "23", "origin": "14294", "source_id": "1893", "source": "cont
ent desc", "pname": "DemographicGender", "pvalue": "F", "context": "application
P:DemographicsPrep", "datapoints": 0}]' -H "Content-Type: application/json" -X
POST "https://spice.fdi.ucm.es/hecht/v2.0/users/23/update-generated-content"
```

4.3.3 Similarity operations

A read-only API endpoint for retrieving information about similarities and dissimilarities between communities.

Usage:

```
GET /communities/{community-id}/similarity
```

Returns a list with the k most similar communities to the chosen one in the community model.

Parameters

k: Size of the result (k most similar communities).

Responses

A list of the similarity scores between the parameter community and the k-most similar communities, in descending order. Every similarity score is represented in an object with the following attributes:

- **target-community-id**: Unique id for the community parameter in the community model
- **other-community-id**: Unique id for another community in the community model
- **value**: Similarity value between the specified communities.
- **similarity-function**: similarity function employed to compute this similarity score.

A response with status 400 is generated if the community model does not contain any community with this community id.

Example request

```
curl -X GET
"https://spice.fdi.ucm.es/hecht/v2.0/communities/621e53cf0aa6aa7517c2afdd/simila
rity?k=5"
```

```
GET /communities/{community-id}/similarity/{other-community-id}
```

Returns the similarity score between two communities.

Parameters

None

Responses

A similarity score object. The similarity score is always 1.0 if both communities are the same.

A response with status 400 is generated if the community model does not contain any community with any of the community ids contained in the URL.

Example request

```
curl -X GET
"https://spice.fdi.ucm.es/hecht/v2.0/communities/621e53cf0aa6aa7517c2afdd/similarity/721e53cf0aa6aa7517c2afdd"
```

GET /communities/{community-id}/dissimilarity

Returns a list with the k most dissimilar communities to the chosen one in the community model.

Parameters

k: Size of the result (k most similar communities).

Responses

A list of the dissimilarity scores between the parameter community and the k-most dissimilar communities, in descending order. Every dissimilarity score is represented in an object with the following attributes:

- **target-community-id**: Unique id for the community parameter in the community model
- **other-community-id**: Unique id for another community in the community model
- **value**: Dissimilarity value between the specified communities.
- **similarity-function**: dissimilarity function employed to compute this dissimilarity score.

A response with status 400 is generated if the community model does not contain any community with this community id.

Example request

```
curl -X GET
"https://spice.fdi.ucm.es/hecht/v2.0/communities/621e53cf0aa6aa7517c2afdd/dissimilarity?k=5"
```

GET /communities/{community-id}/dissimilarity/{other-community-id}

Returns the dissimilarity score between two communities.

Parameters

None

Responses

A dissimilarity score object. The dissimilarity score is always 0.0 if both communities are the same.

A response with status 400 is generated if the community model does not contain any community with any of the community ids contained in the url.

Example request

```
curl -X GET
"https://spice.fdi.ucm.es/hecht/v2.0/communities/621e53cf0aa6aa7517c2afdd/dissimilarity/721e53cf0aa6aa7517c2afdd"
```

4.3.4 Perspective operations

The perspectives define different ways to infer the implicit communities in terms of the citizen contributions and how similar the artworks that citizens interact with are. They are the key concept to configure the community model (see Deliverable 3.7 for more information about perspectives and community model). The Community Model API provides access to configure and retrieve the perspectives.

Usage:

GET `/perspectives`

Returns a list of the perspectives in the community model. If the CM update is necessary returns a job.

Parameters

None

Reponses:

A list of all perspectives in the community model (see next operation)

A response with status 202 returns a job that must be checked until the Community model update finishes.

Example request:

```
curl -X GET "https://spice.fdi.ucm.es/hecht/v2.0/perspectives"
```

GET `/perspectives/{perspectiveId}`

Returns information about a perspective. If the CM update is necessary returns a job.

Parameters

None

Reponses:

An object that describes the perspective configuration in the community model. It contains the following attributes:

- **Id**: Unique id
- **Name**: Perspective name
- **algorithm**: an object with the algorithm configuration used for creating the communities using this perspective.
- **similarity_function**: a list with the similarity functions employed by the community model.
- **user_attributes**: a list with the user attributes employed for defining the explicit communities.

Example request:

```
curl -X GET "https://spice.fdi.ucm.es/hecht/v2.0/perspectives/641987b1a1605e13287486a7"
```

POST `/perspectives/`

Adds a new perspective to the community model and updates it, running the clustering algorithm.

Request Body:

An object that describes the perspective configuration in the community model. It contains the following attributes:

- **Id**: Unique id
- **Name**: Perspective name

- **algorithm**: an object with the algorithm configuration used for creating the communities using this perspective.
- **similarity_function**: a list with the similarity functions employed by the community model.
- **user_attributes**: a list with the user attributes employed for defining the explicit communities.

Parameters

None

Reponses:

A response with status 204 with a job id represents that the perspective was added and the clustering algorithm is running.

A response with status 400 indicates that the perspective configuration object is not correct.

Example request:

```
curl -d '{
"algorithm": {"name": "agglomerative", "params": [], "weight": "0.5", "weightArtworks": "0.5"}, "id": "641987b1a1605e13287486a7", "name": "SimEmotionsSimilarArworkbyIconClass", "similarity_functions": [{"sim_function": {"on_attribute": {"att_name": "iconclassArrayIDs", "att_type": "List"}, "name": "IconClassSimilarityDAO", "params": [], "dis_similar": false}}], "user_attributes": [{"att_name": "demographics.RelationshipWithArt", "att_type": "String"}, {"att_name": "demographics.ContentInLIS", "att_type": "String"}], "interaction_similarity_functions": [{"sim_function": {"on_attribute": {"att_name": "interest.itMakesMeThinkAbout.emotions", "att_type": "dict"}, "interaction_object": {"att_name": "id", "att_type": "String"}, "name": "ExtendedPlutchikEmotionSimilarity", "params": []}}]}' -H "Content-Type: application/json" -X POST
"http://spice.fdi.ucm.es/hecht/v2.0/perspectives/"
```

DELETE /perspectives/{perspectiveId}

Adds a new perspective to the community model and updates it, running the clustering algorithm.

Deletes an existing perspective, referenced by ID.

Parameters:

None.

Responses:

A response with status 200 if the deletion request is performed successfully.

A response with status 400 indicates that there is not a perspective with the provided ID.

A response with status 401 indicates that the user is not authorized to perform that request.

Example request:

```
curl -X DELETE "https://spice.fdi.ucm.es/hecht/v2.0/perspectives/641987b"
```

4.3.5 VISIR operations

VISIR is a visualization tool that helps curators in the interpretation reflection loop process and is employed to configure the perspectives in the community model. See Deliverable D.X.X for more information about VISIR. Community model generates visualization files for the tool and provides specific operations for VISIR:

Usage:

GET /seed

Returns a seed file, a file with metainformation about the data that will be used by the community model. This seed file will be employed by the perspective configuration tool for building new perspectives and add them to the community model.

Parameters

None

Reponses:

A seed file, a file with metainformation about the data in the community model.

Example request:

```
curl -X GET "http://spice.fdi.ucm.es/hecht/v2.0/seed"
```

GET /files

Returns a list of the visualization files generated in the community model. If the CM update is necessary returns a job.

Parameters

None

Reponses:

A list of all visualisation file ids in the community model

A response with status 202 returns a job that must be checked until the Community model update finishes.

Example request:

```
curl -X GET "http://spice.fdi.ucm.es/hecht/v2.0/files"
```

GET /files/{fileId}

Returns a visualisation file. If the CM update is necessary returns a job.

Parameters

None

Reponses:

A visualisation file with information about the communities in a perspective, their user, and the list of interactions needed by VISIR

Example request:

```
curl -X GET "http://spice.fdi.ucm.es/hecht/v2.0/files/641987b1a1605e13287486a7"
```

4.3.6 Development operations

Community model provides some entry points for operations related to the development process and the deployment of community model servers.

Usage:

GET /logs

Returns last N log messages from the community model.

Parameters

- **NLogs** (required): the number of last retrieved log files
- **logsType** (optional): Type of logs ("ALL" "DEBUG" "INFO" "WARNING" "ERROR" "CRITICAL")

Reponses:

A list of N log objects.

A response with status 400 indicates that nLogs is missing or the log type is not correct.

Example request:

```
curl -X GET "http://spice.fdi.ucm.es/hecht/v2.0/logs?nLogs=2"
```

GET /logs/dateRange

Returns log messages between two dates.

Parameters

- **startDate** (required). Starting date in ISO 8601 format: YYYY-MM-DD or YYYY-MM-DDTHH:MM:SS
- **endDate** (required). End date in ISO 8601 format: YYYY-MM-DD or YYYY-MM-DDTHH:MM:SS
- **LogsType**: Type of logs ("ALL" "DEBUG" "INFO" "WARNING" "ERROR" "CRITICAL")

Reponses:

A list of all the log objects generated in the date range.

A response with status 400 indicates that the date is not in correct format or startDate is later than endDate.

Example request:

```
curl -X GET "http://spice.fdi.ucm.es/hecht/v2.0/logs/dateRange?startDate=2023-03-21T00:00:00Z&endDate=2023-03-22T00:00:00Z"
```

GET /database-controller/dump

Returns a dump file in JSON format from the MongoDB database used by the community model.

Parameters

None

Reponses:

A dump of the database

Example request:

```
curl -X GET "http://spice.fdi.ucm.es/hecht/v2.0/database-controller/dump"
```

GET /database-controller/dump

Overwrites the MongoDB database used by the community model with a dump file.

Request body

A dump file of the community model database

Reponses:

The uploaded file if the database was correctly overwritten.

Example request:

```
curl -d @dump.json -X POST "http://spice.fdi.ucm.es/hecht/v2.0/database-controller/dump"
```

4.4 OpenAPI Specification

CM-API uses a dynamically generated OpenAPI specification. A static version of the full OpenAPI 3.0.1 specifications in YAML format has been materialised for the purposes of this report, based on v1.1 of the CM-API. It is available within the CM-API repository in the following file location:

<http://spice.fdi.ucm.es/openapi.yaml>

5 SPICE Semantic Annotator API

5.1 Description of the system

Spice Semantic Annotator (SSA) is an annotation service for the semantic enrichment of textual contents, targeting user generated contents as well as descriptions of museum artefacts. The service is multilingual and supports English, Finnish, Hebrew, Italian and Spanish. It consists of a natural language processing pipeline that performs:

- Sentiment Analysis,
- Emotion Detection
- Entity Linking
- Hate Speech Detection

The process of semantic annotation is realized by a **Natural Language Processing Pipeline** that includes different analysis modules, each one responsible for annotating the document with respect to a specific aspect: sentiment analysis, emotion detection, entity linking. The overall process is exposed by means of standard **RESTful¹ APIs** and produces a JSON-LD² document as output. **JSON-LD** is a JSON-based serialization for Linked Data that can be seamlessly stored in the Linked Data hub of WP4.

SSA analyses textual contents collected from museum visitors interacting with the activities scripted in the interfaces (WP5) and realized for the different use cases (WP7). The service annotates contents with respect to the ontological models developed in WP6 and generates as output an RDF graph to be stored in the linked data hub developed by WP4. Such analysis puts the visitor at the centre by interpreting and then enhancing his point of view and contributes to:

- the process of defining profiles of each visitor in order to build Community Models (the profiles and models are generated by task 3.1).
- the design of an advanced recommendation engine (task 3.3)
- The pipeline is designed following a **Microservice Architecture³** approach, exploiting a wide variety of models and solutions available on the open source.
 - It is deployed as a Microservice Architecture on a Kubernetes⁴ cluster. **Kubernetes** is an open-source system for automating deployment, scaling, and management of containerized components (e.g., Docker⁶ images).

The whole system is deployed to AWS⁷ cloud resources, on servers located in the European region. The architecture and layout of the service is provided below in Figure 5.1.1.

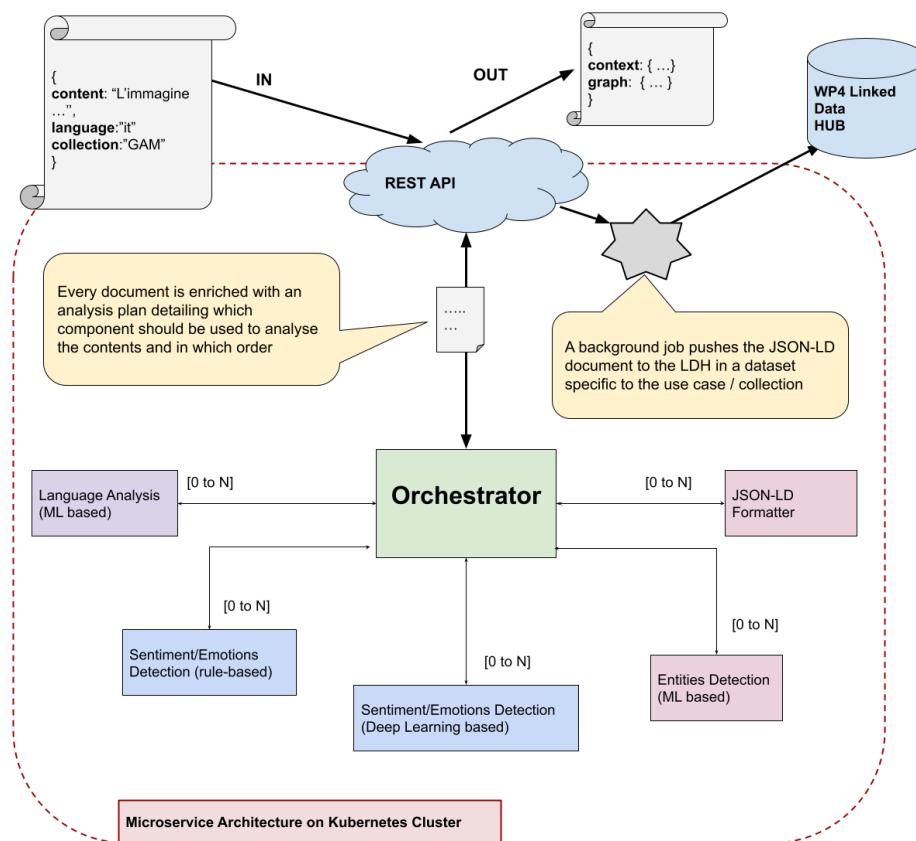


Figure 5.1.1. SPICE Semantic Annotator Architecture

This section describes SSA API detailing about its input, output and usage. The service is exposed through standard REST API behind a Basic Authentication¹⁸ scheme. The service can be accessed at the URL:

- <https://analytics-demo.aws.celi.it/<LANGCODE>/spice/analysis>

<LANGCODE> is a path parameter and it is used to specify the language content, the supported values are:

- `en,es,fi,he,it`

The service can be activated on request (by contacting sophiaanalytics.support@h-farm.com for the activation and the authentication details) and can be accessed through POST requests, accepting a json document as input, with the following properties:

- ∉ `content`: *mandatory* - the textual contents to be analyzed
- ∉ `ns_prefix`: *optional* - the prefix used for representing the textual content in the JSON-LD response document, default value is "`spice`"
- ∉ `ns_uri`: *optional* - the URI of the ontology used for representing the textual contents in the JSON-LD document, default value is "`https://w3id.org/spice/resource/`"
- ∉ `collection`: *optional* - a textual label representing the collection/museum/use case, default value is "`spice`"

An example of input is:

- `{"content": "I love this painting!", "collection": "test"}`

5.2 Metadata

id	SSA
name	SPICE Semantic Annotator API
description	Final Version of SSA
type	Rest API
release-date	14/01/2023
release-number	1.2
work-package	WP3
keywords	Semantic annotation of User Generated Contents
credits	Alessio Bosca (alessio.bosca@h-farm.com)

5.3 Guide for developers

An example API request to SSA service API, using python with the well known requests²¹ lib:

```
import requests

def testService(text: str, lang: str) -> object:
    r = requests.post(' https://analytics-demo.aws.celi.it/'+lang+'/spice/analysis',
                      json={"content":text, "collection":"test"},
                      auth=('USR', 'PWD'))
    print(r.json())

if __name__ == "__main__":
    testService("I love Picasso's Guernica, but I am absolutely terrified by the screaming horse!", 'en')
```

SSA API request example via Python

Please notice that USR and PWD **MUST** be substituted with a real authentication in order to access the API.

The Semantic Annotator exposes the NLP pipeline analysis results as a JSON-LD²² document. JSON-LD is a method of encoding linked data using JSON. Linked Data is structured data which is interlinked with other data so it becomes more useful through semantic queries. It builds upon standard Web technologies such as HTTP, RDF and URIs. More details on the Linked Data Hub designed and deployed by WP4 can be found in **D4.1 Linked Data server technology: requirements and initial prototype**.

The JSON-LD document contains two main sections:

- ≠ **Context:** detailing the ontologies used to describe data along with their prefix (used for compact notations in the graph section)
- ≠ **Graph:** containing a set of RDF triples represented as JSON objects; in our case the textual contents along with some metadata, followed by a set of annotations referencing the textual spans that can be linked to an emotion, a sentiment value or an entity (within DBpedia knowledge graph)

The following image represents the service output for the input: “I love Picasso's Guernica, but I am absolutely terrified by the screaming horse!”

```
{
  "@context": {
    "spice": "https://w3id.org/spice/resource/",
    "owl": "http://www.w3.org/2002/07/owl#",
    "dbr": "http://dbpedia.org/resource/",
    "earmark": "http://www.essepuntato.it/2008/12/earmark#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "dcterms": "http://purl.org/dc/terms/",
    "semiotics": "http://ontologydesignpatterns.org/cp/owl/semiotics.owl#",
    "emotion": "https://w3id.org/spice/SON/PlutchikEmotion/",
    "marl": "http://www.gsi.upm.es/ontologies/marl/ns#",
    "sentilo": "http://ontologydesignpatterns.org/ont/sentilo.owl"
  },
  "@graph": [
  ]
}
```

Figure 6.3.1. SSA JSON-LD output - @context section

In the LDH, a specific dataset for each museum is used to collect all users’ generated content related to a specific use case. One of the parameters of SSA API consists of a label for the collection of the contents to be analysed. If the value of the collection parameter refers to one of the museum use cases, then the JSON-LD document is saved in a use case specific dataset, otherwise a fallback test dataset is used.

The following table details the museum specific collections along with the relative dataset UUID; the fallback test dataset details are also reported at the end of the table.

Collection - Museum Use Case	Dataset UUID in LDH
IMMA	b3631f48-2657-4cd3-96fa-4887c6e0c63a
GAM	810d60a6-c7be-4299-be2e-c86d988f58ad
HECHT	4125ba0c-adbe-4b0b-a2ff-3a5dde29d088
MNCN	2ae73c0c-84ad-416c-b17b-23032a75f0ef
DMH	514c5676-2560-47a9-bab4-76ff42eb0b83
test	85c109bb-6090-4110-9422-79303183fae5

6 Social Recommender API

6.1 Description of the system

The purpose of this component is to provide social recommendations based on user generated content to aid in the implementation of the interpretation-reflection loop of WP2; primarily reflection. You can choose similar viewpoints by different communities to engender inclusion and use different viewpoints by similar communities to try and engender cohesion. The recommendations are based on similar and dissimilar views of topics and subjects and material from both similar and dissimilar communities. Views of the subject are collected by the Semantic Analyzer. These are stored in the UM for use by the CM which generates similar and dissimilar communities which is then used by the SR to provide recommendations.

The API consists of a single call which attempts to do as much as possible for the developer in providing recommendations of user generated content. Communities can either be explicit or implicit

The idea is to give the script designer the possibility to find people who have a common background but have either a different or same opinion or alternatively people who think alike but have a different or same opinion

A typical scenario is that a recommendation of user generated content (opinion, curation and so on) is requested. User generated content is chosen based on the analysis of the Semantic Analyser (similar or dissimilar). It is then filtered by content belonging to users of certain communities.

A prototype demo is incorporated in the *Hecht studentmgr* demo.

For more detailed information See D3.8.

6.2 Metadata

id	SR-API
name	Social Recommender API
description	Provide similar/dissimilar recommendations bases on similar/dissimilar communities
type	API
release-date	1-5-2023
release-number	1.0
work-package	WP3
pilot	Hecht studentmgr
keywords	Social Recommender, De-polarization
licence	Apache
demo	Hecht studentmgr
credits	Alan J. Wecker (ajwecker@gmail.com) Tsvika Kuflik (tsvikak@is.haifa.ac.il)
bibliography	See D3.8

6.3 Guide for developers

The API is called using a standard Spring Boot REST API call.

Given several parameters, it returns a list of User Generated Content items (UGC).

Recommendation-controller

[GET /api/v2/srec/{userid}/{subject}/{configNum}/{criterionInput}](#)

Get recommendation for user

Parameters

Name	Description
userid * string <i>(path)</i>	Anonymous id
subject * string <i>(path)</i>	Topic of interest or artwork
configNum * string <i>(path)</i>	1 – similar criterion similar community 2 – dissimilar criterion similar community 3 – similar criterion dissimilar community 4 – dissimilar criterion dissimilar community
criterionInput * string <i>(path)</i>	sentiment or emotion

Responses

Code	Description

200	<p>OK</p> <p>Media type</p> <p>*/*</p> <p>Controls Accept header.</p> <ul style="list-style-type: none"> • Example Value • Schema (Recommendation) <pre> { "entrancement": "string", "explanation": "string", "ugcs": [{ "source": "string", "context": "string", "createdAt": "string", "updatedAt": "string", "_id": "string", "_docType": "string", //SPICEUMUGC "userid": "string", "parentname": "string", "parenttype": "string", "contentType": "string", "ugcname": "string", "ugcimage": "string", "ugctags": "string", "ugcdesc": "string", "ugcmeta": [/raw results from SSA {}], "emotions": {}, //result from SSA "sentiments": {}, //result from SSA "entities": {}, //result from SSA "ugcmeta2": "string", //result from values "topic": "string", "ugctext": "string //Text without markup }] } </pre>
-----	--

6.4 API Schemas

Recommendation

entrancement	String {personalized enhancement why you might want to view this recommendation}
explanation	String {Why was this recommendation given}
ugcs	[...] List of recommended UserGeneratedContent

7 Ontology server, query and reasoning services

In this Chapter, we describe the Ontology server developed in WP6 in detail. While the previous chapters describe software component already described in other deliverables, here we present the ontology server as both system specification and running instance.

As detailed in Deliverable 6.4 the APIs for the ontology architecture relies on the APIs of the a number of integrated services. In particular: the ontology server in the SPICE technical infrastructure runs via a virtual machine instance, within the HPC4AI cloud infrastructure of the Department of Computer Science of the University of Turin (<https://hpc4ai.unito.it/>), and managed through the OpenStack console (<https://www.openstack.org/>).

The server has a total volume storage of 500GB, a RAM of 256GB, an availability of 8VCPUs (currently the SPICE virtual machine uses 4 VCPUs) and is equipped with the Ubuntu 18.04 operating system. It has an external IP address reachable by other services via HTTP. The SPICE Server hosts and integrates the following software components:

- OWL-API 5 (<http://owlcs.github.io/owlapi/>), a Java API and reference implementation for creating, manipulating, serialising OWL Ontologies and using OWL Reasoners;
- HERMIT (<http://www.hermit-reasoner.com/>), a standard ontology-based reasoner used to infer taxonomical and hidden relationships between elements from a knowledge base. The reasoner is called externally via the OWL-API;
- ONT-API 2.0.0 (<https://github.com/owlcs/ont-api>), a java library for converting the OWL and OWL 2 ontologies (with the materialized inferences obtained through reasoning) in RDF-like graph models stored as triples;
- JENA (<https://jena.apache.org/index.html>), an Apache framework for storing, manipulating and accessing RDF graphs.
- Fuseki 2 (<https://jena.apache.org/documentation/fuseki2/index.html>) a SPARQL server of the Apache family used to expose the Jena models (containing the inferred triples) to other services both via a public interface (available at this address: <http://130.192.212.225/fuseki/>) and via SOH (SPARQL Over HTTP): a set of server-independent command-line scripts for working with SPARQL 1.1 offering HTTP access to external services. Fuseki 2 is hosted in a Tomcat Server to be exposed and reachable on the Web. The overall architecture is illustrated in the Figure 7.0.1 below.

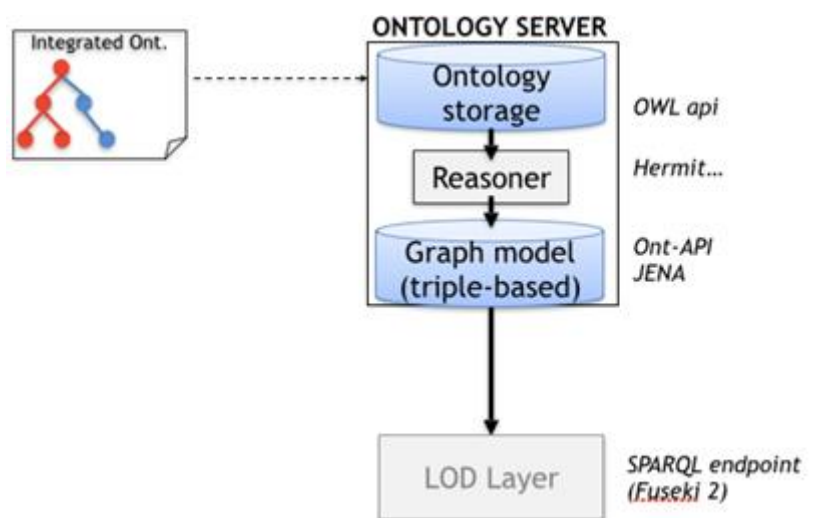


Figure 7.0.1. Overview of the components of the Ontology Server and Reasoning Architecture

This architecture implements the following workflow:

1. An OWL/OWL 2 ontology (or a network of ontologies) is uploaded to the Ontology Server via OWL-API¹.
2. All the triples that can be deduced by the loaded ontology, which form the so called “inferred ontology” are then derived by using a standard OWL reasoner (e.g., Hermit)².
3. The inferred ontology is then transformed to graph model (which is the format suitable for enabling querying via SPARQL). This is done by relying on the ONT-API framework, which transforms the inferred model into graph model compatible with Jena and Fuseki frameworks.
4. The model is automatically loaded in the Fuseki 2 SPARQL server available for querying at <http://130.192.212.225/fuseki/>

We will detail below, step by step, how the different components are integrated by using a running example about a toy knowledge base (called “arte”) loaded, reasoned and exposed in a SPARQL endpoint as a turtle file (.ttl extension). The example will be discussed by providing the different pieces of code necessary to activate the different components (the overall code provided in the example is available in the final appendix).

7.1 Ontology Uploading and Reasoning calls via OWL-API (Steps 1-2)

An ontology, or a network of ontologies, can be uploaded to our ontology server via OWL-API as a file, or by using an external IRI pointing to an OWL ontology. Since the uploading procedure via file can only be done by the managers of the infrastructure (i.e., UNITO members) we have opted for the upload via an external IRI, that is reachable and directly usable by any user of the project. The IRI from which it is possible to upload the ontologies is: <http://130.192.212.225/fuseki> (the public address of the Fuseki 2 repository).

In the code shown below in Figure 7.1.1, we show how an ontology stored at the IRI <http://130.192.212.225/fuseki/arte> (where “arte” is the name of the newly created dataset in Fuseki 2 storing the “arte.ttl” ontology) is loaded via OWL-API and how an ontological reasoner (e.g., HERMIT) is initialized by indicating the types of inferences we are interested in (e.g., CLASS_HIERARCHY, CLASS_ASSERTIONS, DIFFERENT_INDIVIDUALS etc.).

```
//Ontology loading (OWL-API and ONT-API):
OWLOntologyManager man = OntManagers.createONT();
IRI arteIRI =
IRI.create("http://130.192.212.225/fuseki/arte");
OWLOntology o = man.loadOntology(arteIRI);

//Reasoning calls (Hermit):

OWLReasonerFactory rf = new ReasonerFactory();
OWLReasoner r = rf.createReasoner(o);
r.precomputeInferences(InferenceType.CLASS_HIERARCHY);
r.precomputeInferences(InferenceType.CLASS_ASSERTIONS);
r.precomputeInferences(InferenceType.DISJOINT_CLASSES);
r.precomputeInferences(InferenceType.DIFFERENT_INDIVIDUALS);
r.precomputeInferences(InferenceType.OBJECT_PROPERTY_ASSERTIONS);
```

Figure 7.1.1. Code excerpt for ontology uploading via external IRI and initialization of the reasoning procedures

7.2 Ontology Export as JENA Triple-based graph model (step 3)

In order to expose the reasoned ontologies in a format that is also accessible via SPARQL queries, it is necessary to translate the OWL / OWL 2 ontology/ontologies in a JENA graph model. This service is provided by the ONT-API library. In order to activate this translation, the following instruction (to be intended as a continuation of the code illustrated in Figure 7.1.1) is provided:

```
Model model = ((Ontology(o)).asGraphModel());
```

It allows to generate and store the JENA Graph Model as a JAVA object. It is worth noticing that the Model could also eventually be written into a file by providing this additional instruction:

```
model.write(FileOutputStream(new File(ReasonedArte.ttl)), "ttl").
```

This possibility, however, is not currently used because it does not allow to automatically upload the JENA model to the external SPARQL server (that is, on the other hand, what we aim to do).

7.3 External Exposure of the Graph Model in a SPARQL Server (step 4)

We upload the extracted JENA model to a Fuseki 2 SPARQL server. In Fuseki 2 it is possible to expose the JENA models as different datasets in two different ways: in a manual way and in an automatic fashion.

The manual upload can be done by using the **SOH function (SPARQL over HTTP functions)**³ provided by Fuseki 2 or by manually uploading the file on the Fuseki 2 SPARQL graphical interface. SOH provides a set of Ruby scripts runnable from command line (therefore it is necessary to install Ruby in order to run them).

In our example, for what concern the use of the “UPLOAD” SOH command it is sufficient to provide the following PUT script to upload the file “ReasonedArte.ttl”:

```
s-put http://130.192.212.225/fuseki/arte default ReasonedArte.ttl
```

Alternatively, the file can be simply uploaded via the SPARQL graphical interface (see Figure 7.3.1), reachable at <http://130.192.212.225/fuseki/arte>



Figure 7.3.1. SPARQL endpoint interface for the upload

As mentioned above, however, we decided to opt for a completely automatic upload of the JENA model Graph Model (containing the inferred ontology) in Fuseki 2 (at the scope of making it available such model via a reachable SPARQL endpoint).

In order to do that, we used the RDF Connection component of JENA API³. This process is obtained by the following instruction (in bold the new instructions with respect to the previous ones):

```

OWLOntologyManager man = OntManagers.createONT();
IRI arteIRI = IRI.create("http://130.192.212.225/fuseki/arte");
OWLOntology o = man.loadOntology(arteIRI);

/*
 * Reasoning/manipulation of the OWL model
 */

Model model = ((Ontology)o).asGraphModel();

RDFConnectionRemoteBuilder builder =
RDFConnectionFuseki.create()
.destination("http://130.192.212.225/fuseki/arte");

try ( RDFConnectionFuseki conn =
(RDFConnectionFuseki)builder.build() ) {
    conn.put(model);
}
    
```

Figure 7.3.2. Automated export of the JENA Graph Model in the Fuseki 2 SPARQL Server via JENA API

This overall workflow allowed us to completely automate the connections between the ontology server and the SPARQL one. As already mentioned, we opted for an upload of the ontology via OWL-API through an external IRI. The external IRI provided to OWL-API resolves to the Fuseki repository reachable (in our example) at the address <http://130.192.212.225/fuseki/arte>. Once the ontology is loaded in OWL-API, it goes through the whole processes of reasoning, translation to JENA model, and automatic update of the Fuseki 2 repository without any manual intervention.

In the following we describe how to query and update (including inserts, deletes, updates) the Fuseki 2 exposed model by means of the services: SOH and RDF Connection. Query and update are also available through the graphical interface.

7.4 How to QUERY and UPDATE the exposed Fuseki 2 Model with SOH

In order to query the exposed Fuseki 2 Model via SOH, it is sufficient to use the scripting instructions provided here: <https://jena.apache.org/documentation/fuseki2/soh.html>.

For example: a simple query could be run via command line in the following way (see Figure 3.4.1 below).

```

./s-query --service=http://130.192.212.225/fuseki/arte 'SELECT ?s ?p ?o W
HERE { ?s ?p ?o . }'
    
```

³ <https://jena.apache.org/documentation/rdfconnection/>

```

tomcat@spice-server:~/apache-jena-fuseki-3.16.0/bin$ ./s-query --service=http://130.192.212.225/fuseki/a
rte 'SELECT ?s ?p ?o WHERE {?s ?p ?o}'
{ "head": {
  "vars": [ "s", "p", "o" ]
},
  "results": {
    "bindings": [
      {
        "s": { "type": "uri", "value": "http://www.modsem.org/arte#OperaContemporanea" },
        "p": { "type": "uri", "value": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" },
        "o": { "type": "uri", "value": "http://www.w3.org/2002/07/owl#Class" }
      },
      {
        "s": { "type": "uri", "value": "http://www.modsem.org/arte#OperaContemporanea" },
        "p": { "type": "uri", "value": "http://www.w3.org/2000/01/rdf-schema#subClassOf" },
        "o": { "type": "uri", "value": "http://www.modsem.org/arte#Opera" }
      },
      {
        "s": { "type": "uri", "value": "http://www.modsem.org/arte#OperaContemporanea" },
        "p": { "type": "uri", "value": "http://www.w3.org/2002/07/owl#equivalentClass" },
        "o": { "type": "bnode", "value": "b0" }
      },
      {
        "s": { "type": "uri", "value": "http://www.modsem.org/arte#Opera" },
        "p": { "type": "uri", "value": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" },
        "o": { "type": "uri", "value": "http://www.w3.org/2002/07/owl#Class" }
      },
      {
        "s": { "type": "bnode", "value": "b0" },
        "p": { "type": "uri", "value": "http://www.w3.org/1999/02/22-rdf-syntax-ns#type" },
        "o": { "type": "uri", "value": "http://www.w3.org/2002/07/owl#Restriction" }
      }
    ]
  }
}
    
```

Figure 7.4.1. Query of the FUSEKI 2 server via terminal with the SOH scripts.

By using SOH it is also possible to upload/modify the Fuseki 2 model via command line. For example, it is possible to include additional information in our toy model about the artistic domain in the following way:

```

s-update --
service=http://130.192.212.225/fuseki/arte

'PREFIX : <http://www.modsem.org/arte#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX xsd:
<http://www.w3.org/2001/XMLSchema#>

INSERT DATA {

    :LaVita rdf:type owl:NamedIndividual,
            :Dipinto ;
    :haTitolo "La Vita"^^xsd:string .

    :PabloPicasso :creatoreDi :LaVita .
}
    
```

Figure 7.4.2. This simple instruction allows adding that “La Vita” (La Vie) is a painting by Picasso.

7.5 How to QUERY and UPDATE the exposed Fuseki 2 Model with RDF Connection

In order to use **RDF Connection** for both SPARQL query and Model Updates, we need to first establish a connection between the Fuseki 2 Server and the corresponding Model that we want to query/update. With the RDFConnection interface, SPARQL operations can be performed on Fuseki remote datasets.

In order to activate an RDF Connection, it is necessary to include the instruction mentioned above and used for the automatic upload of the JENA graph-model in Fuseki 2. The remote connection to the “arte” dataset is provided as follows:

```
RDFConnectionRemoteBuilder builder = RDFConnectionFuseki.create()
    .destination("http://130.192.212.225/fuseki/arte");
```

Once the connection is established, it is possible to use RDF connection to provide SPARQL Queries and Model Updates on the Fuseki 2 Model.

For example: a query about the paintings of Picasso contained in the “arte” knowledge base can be executed programmatically in this way:

```
Query query = QueryFactory.create(
    "PREFIX arte:
    <http://www.modsem.org/arte#> " +
    "SELECT ?opera " +
    "WHERE { arte:PabloPicasso
    arte:creatoreDi ?opera .}");

try ( RDFConnectionFuseki conn =
    (RDFConnectionFuseki)builder.build() ) {
    conn.queryResultSet(query,
    ResultSetFormatter.out);
}
```

Figure 7.5.1. SPARQL query execution on Fuseki 2 from an external application via RDF Connection

Similarly, the knowledge base can be updated by adding new information in this simple way:

```
try ( RDFConnectionFuseki conn =
    (RDFConnectionFuseki)builder.build() ) {
    conn.update(
    "PREFIX : <http://www.modsem.org/arte#> " +
    "PREFIX owl: <http://www.w3.org/2002/07/owl#> " +
    "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> "
    +
    "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> " +
    "INSERT DATA { " +
    "    :LaVita rdf:type owl:NamedIndividual, " +
    "    :Dipinto ; " +
    "    :haTitolo \"La Vita\"^^xsd:string . " +
    "    :PabloPicasso :creatoreDi :LaVita .\n" +
    "});
    conn.queryResultSet(query, ResultSetFormatter.out);
}
```

Figure 7.5.2. Knowledge update on Fuseki 2 from an external application via RDF Connection

Some insertions or deletions could be not compliant with the dictate of the ontological models. In case some inconsistent information is added, the OWL-API component (that loads the ontology from the IRI we are manipulating) can detect, using the launched reasoner, that the included information is inconsistent and should be modified or deleted.

7.6 An overview of the API and workflow architecture for DEGARI GAMStories

Figure 7.6.2 is an example of a story entitled "The course of nature" created by the GAMGame user and the associated Plutchik's wheel for the extracted emotions. The Figure 7.6.3 shows an example of RDF-SPARQL query on Fuseki server. Given a story-ID, this query select the sets of the stories which have the opposite emotions with the current story. In particular, in this example, it is possible to see that Figure 2.5 shows an example of RDF-SPARQL query for retrieving stories which have same emotions with the current one. In the Table 7.6.1 is reported an example of a story. In particular, user with id="6rK3p7za" creates a story entitled "Sad sea", selecting three different artworks 39138,35362, 35249. Figure 2.6 shows the Plutchik's wheel extracted from the story "Sad sea". The emotional wheel was calculated by averaging the i-emotions extracted from the union of each artworks that characterise the whole story.

StoryID	UserID	StoryTitle	ArtefactID	Artefact name
6333f37613a6b278b06dc2e7	6rK3p7za	Sad sea	39138	(Der) Matrose Fritz Müller aus Pieschen
			35362	Sheds by the sea (Capanni sul mare)
			35249	Composition T. 50 - 5

Table 7.6.1. Story example



Figure 7.6.2. Example of user story with Plutchik's wheel

**RDF-SPARQL Query
On Fuseki server**

```

11 PREFIX cont: <https://w3id.org/spice/SOIR/emotionInCulturalContext/>
12
13
14 SELECT ?id ?label ?emotion
15 WHERE {
16   story:6239d6a43e14a72415517d5b emo:triggers ?emotion_1;
17   arco:hasConstituent ?c1.
18   ?id rdfs:label ?label;
19   arco:hasConstituent ?c2;
20   emo:triggers ?emotion.
21   ?emotion_1 cont:hasOppositeEmotion ?emotion.
22 }
23
24
    
```

ID	Label	Emotion
34	story:632178935d6a786add2e128e	*Terroro alla GAM*
35	story:632178935d6a786add2e128e	*Terroro alla GAM*
36	story:632178935d6a786add2e128e	*Terroro alla GAM*

Results

emo:Aggressiveness

Figure 7.6.3. Example of RDF-SPARQL API query on Fuseki server for retrieving stories with opposite emotions

7.7 The workflow architecture for DEGARI GAMStories

As we will show in the previous sections, this advancement allowed us to call the DEGARI 2.0 reasoning services to integrate its output within GAMGame, developed to collect stories and user data on cultural items, during a museum visit. Finally, by using Visir, curators in the museum are available to create and visualise different perspectives in order to evaluate different communities formed based on citizens' selections and emotional responses. The pipeline of the GAMGame, DEGARI 2.0 and Visir services is sketched in Figure 7.7.1 and relies on the following workflow:

- (1) Users by using GAMGame web app can create a story and send a JSON file (in JSON-LD format) by using POST method. This JSON file contains the description of a particular story and annotations collected by the users over the artefact (e.g. tags about the emotions generated, emojis etc.). All these annotations will be used as a description of the cultural item under consideration in the following steps
- (2) The JSON-LD file with the ID of the story and its description/items and comments is stored into the Linked data Hub (LDH) and into knowledge graph (KG) stored on Fuseki server
- (3) The algorithms running on the DEGARI server periodically download the data both from the LDH and Fuseki Server (phase 3 and 5 in the Figure 6). They execute the annotations process both with DEGARI, eMFD (extended Moral Foundation Dictionary) and the available eMFD API and ICONCLASS API for the annotation process of materials, technique, colour and objects that characterise artworks
- (4) DEGARI server executes the reclassification of the JSON-LD artefact and sent back to the LDH data containing the enriched stories (emotions + values + ICONCLASS)
- (5) DEGARI algorithms running on server, communicates with a Fuseki server hosting the Plutchik's ontology by automatically updating the knowledge base with the RDF-triples associating to each item a specific emotion. The update is done by using the RDF connection SPARQL Update method provided by the SPARQL APIs. Finally, DEGARI algorithms downloads the new update KG and start reasoning in local by using Hermit reasoning algorithm in order to get the inferences on KG and update the new version of KG (now reasoned) on Fuski server.

- (6) DEGARI server sends all JSON-LD data (in the form of User and Community models) to User Model server, able to store citizen's data with the enriched contribution of previously created stories (UM-data).
- (7) User Model server (UM-server) send UM-data to the Community Model server (CM-server) that is able to clusterize citizens and discover implicit communities by using perspectives, clustering algorithms (with a particular similarity function) and finally shows all the results in an explicable way.
- (8-9) The CM-server sends clustered data to Visir. Visir can support the Interpretation-Reflection-Loop (IRL) by executing the perspective visualisation, communities and users' introspection, inter-intra community visualisation and perspective configuration. All these steps are driven by the domain expert (museum's curators, phase 9 in Figure 6).
- (10) Finally, in the last step, by using the GAMGame, it is possible to perform the task of story recommendation, executing RDF-SPARQL queries directly on the KG stored on Fuseki server.

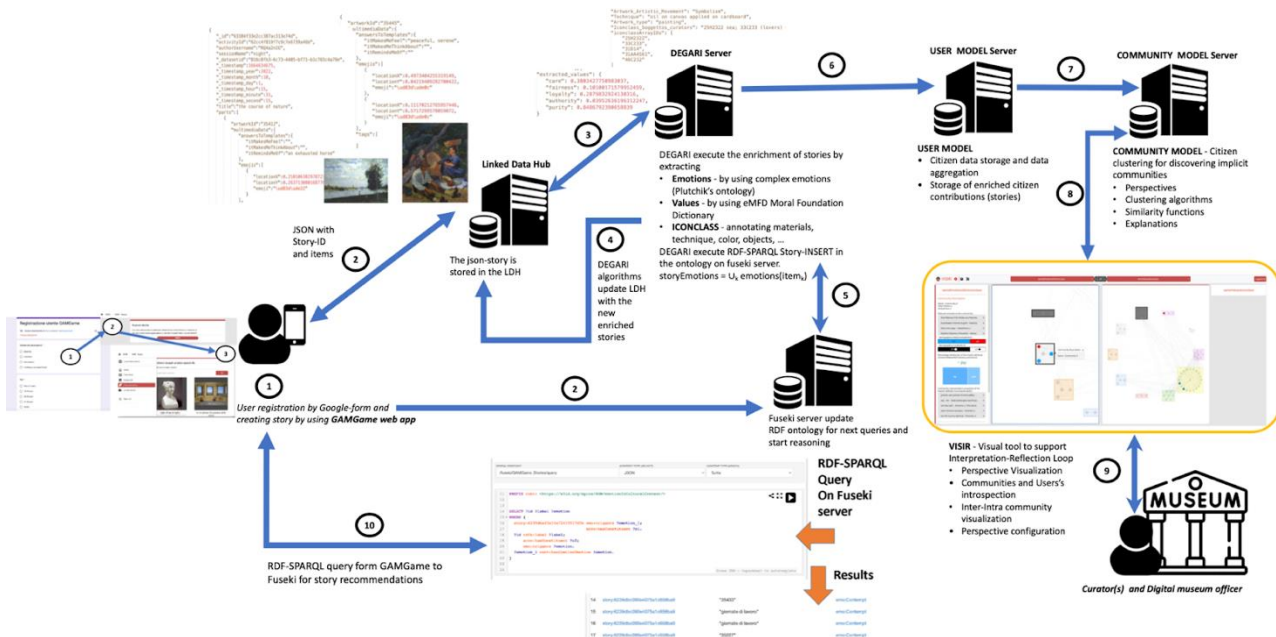


Figure 7.7.1. DEGARI 2.0 API workflow architecture for GAM stories Recommendation

7.8 Insert and update Fuseki with GAM stories

In order to use RDF Connection for both SPARQL query and Model Updates we need to first establish a connection between the Fuseki 2 Server and the corresponding Model that we want to query/update. In particular, via RDFConnection interface, SPARQL operations can be performed on Fuseki remote datasets.

```
String GAMStories_KB = "GAMGame_Stories";
String serverDestination = "http://130.192.212.225/fuseki/" + GAMStories_KB;
```

```
idArtefact = String.valueOf(idArtefact.substring(0, 1).toUpperCase()) + idArtefact.substring(1).toLowerCase();
final RDFConnectionRemoteBuilder builder = RDFConnectionFuseki.create().destination(serverDestination);
```

Once this connection is established it is possible to use RDF connection to provide SPARQL Query, in order to update and adding new information about the triple **<storyId, storyTitle, idArtefact>**. The RDF-SPARQL query is automatically called both from GAMGame (phase 10 in Figure 7.7.1) and DEGARI 2.0 (phase 5 in Figure 7.7.1). An example is shown in Figure 7.7.2.

7.9 Dataset stored in the Linked Data Hub (LDH)

Both GAMGame and DEGARI 2.0 store and download data in JSON-LD format, to and from the Linked Data Hub (LDH). Below is the list with the datasets used by DEGARI 2.0 (only regarding the GAMGame) to implement the entire pipeline implemented in Figure 7.7.1.

This dataset contains all stories created by users using GAMGame web application

- **URL:** <https://spice.kmi.open.ac.uk/dataset/stream/details/58>
- **UUID:** 816c8fb3-6c73-4405-bf71-b1c765c4a79e
- **Name:** Gam_Game_Story_Definitions

This dataset contains the set of 463 artworks of GAM Museum, used by the GAMGame web application. For a small subset of artworks, there are 3 links to YouTube videos, containing the presentation of the artwork in LIS (Italian Sign Language), ITA and ENG.

- **URL:** <https://spice.kmi.open.ac.uk/dataset/details/62#>
- **UUID:** 745d08a3-8f25-4c1a-a34c-e828b7e376d6
- **Name:** GAM_dataset

This dataset contains the set of artworks of GAM Museum, annotated with ICONCLASS, information on artists' art movements, emotions extracted from DEGARI, materials, etc. (phase 4 in Figure 7.7.1)

- **URL:** <https://spice.kmi.open.ac.uk/dataset/json/details/139>
- **UUID:** 2a2a5c9a-a8ce-4977-ba09-f4134c95d744
- **Name:** GAM_Catalogue_plus

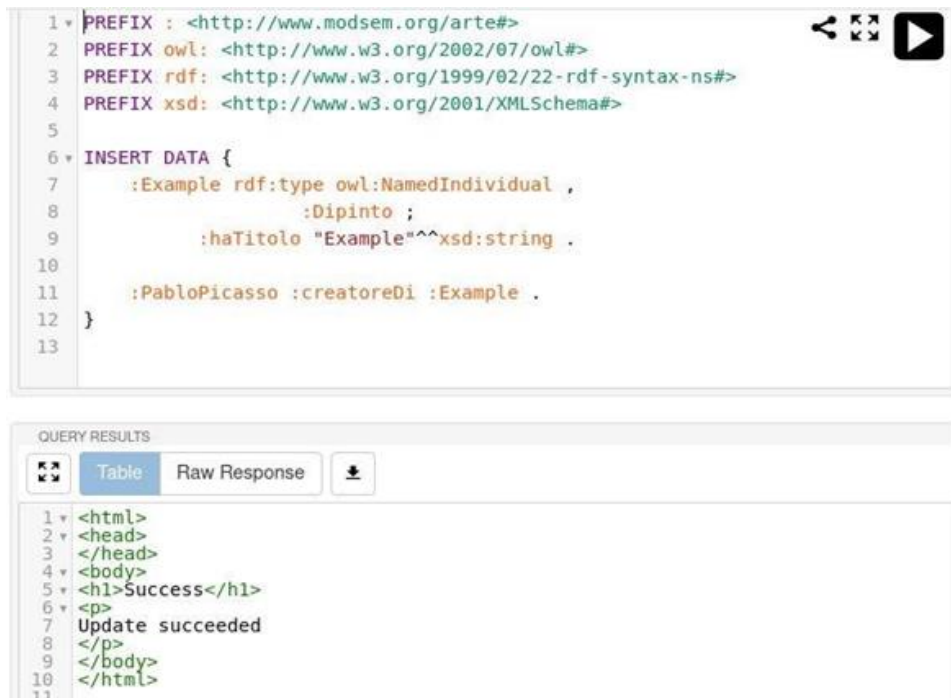
This dataset contains collected information on GAMGame registered users (phase 1 in Figure 7.7.1)

- **URL:** <https://spice.kmi.open.ac.uk/dataset/details/122>
- **UUID:** 495778c1-2509-4a9c-be15-fb0b2e9afc08
- **Name:** GAMGame - information on registered users

7.10 Graphical Interface

The same query and update/delete commands described in the SOH and the RDF Connection can be run directly on the graphical interface of the Fuseki 2 SPARQL endpoint: <http://130.192.212.225/fuseki> (by selecting the repository “arte”).

For example, it is possible to update the Fuseki Model (e.g., by adding an additional painting by Picasso called “Example”), as exemplified in Figure 7.6.1 below.



The screenshot displays the Fuseki 2 graphical interface. The top section shows a SPARQL query editor with the following content:

```

1 PREFIX : <http://www.modsem.org/arte#>
2 PREFIX owl: <http://www.w3.org/2002/07/owl#>
3 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
4 PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
5
6 INSERT DATA {
7   :Example rdf:type owl:NamedIndividual ,
8             :Dipinto ;
9   :haTitolo "Example"^^xsd:string .
10
11   :PabloPicasso :creatoreDi :Example .
12 }
13
    
```

The bottom section, titled "QUERY RESULTS", shows the execution outcome. It includes a "Table" button and a "Raw Response" button. The raw response is displayed as follows:

```

1 <html>
2 <head>
3 </head>
4 <body>
5 <h1>Success</h1>
6 <p>
7 Update succeeded
8 </p>
9 </body>
10 </html>
11
    
```

Figure 7.6.1. Model update in Fuseki 2 via the graphical interface

7.11 Source code

Here we show an excerpt of the code illustrated in the running example, available at <https://github.com/spice-h2020/SPICE-OntoServer>.

```
//Ontology loading (OWL-API and ONT-API):
OWLOntologyManager man = OntManagers.createONT();
IRI arteIRI = IRI.create("http://130.192.212.225/fuseki/arte");
OWLOntology o = man.loadOntology(arteIRI);

//Reasoning calls (Hermit):
OWLReasonerFactory rf = new ReasonerFactory();
OWLReasoner r = rf.createReasoner(o);
r.precomputeInferences(InferenceType.CLASS_HIERARCHY);
r.precomputeInferences(InferenceType.CLASS_ASSERTIONS);
r.precomputeInferences(InferenceType.DISJOINT_CLASSES);
r.precomputeInferences(InferenceType.DIFFERENT_INDIVIDUALS);
r.precomputeInferences(InferenceType.OBJECT_PROPERTY_ASSERTIONS);

//Translation into Jena Model (Apache Jena and ONT-API):
Model model = ((Ontology)o).asGraphModel();

//SPARQL Graph Store Protocol "PUT" on remote Fuseki server (Jena RDF Connection)
RDFConnectionRemoteBuilder builder = RDFConnectionFuseki.create()
    .destination("http://130.192.212.225/fuseki/arte");
try (RDFConnectionFuseki conn = (RDFConnectionFuseki)builder.build() ) {
    conn.put(model);
}

//Example of SPARQL Query on remote Fuseki server (Jena RDF Connection)
RDFConnectionRemoteBuilder builder = RDFConnectionFuseki.create()
    .destination("http://130.192.212.225/fuseki/arte");
Query query = QueryFactory.create(
    "PREFIX arte: <http://www.modsem.org/arte#> " +
    "SELECT ?opera " +
    "WHERE { arte:PabloPicasso arte:creatoreDi ?opera .}");
try (RDFConnectionFuseki conn = (RDFConnectionFuseki)builder.build() ) {
    conn.queryResultSet(query, ResultSetFormatter.out);
}

//Example of SPARQL Update on remote Fuseki server (Jena RDF Connection)
RDFConnectionRemoteBuilder builder = RDFConnectionFuseki.create()
    .destination("http://130.192.212.225/fuseki/arte");
try (RDFConnectionFuseki conn = (RDFConnectionFuseki)builder.build() ) {
    conn.update(
        "PREFIX : <http://www.modsem.org/arte#> " +
        "PREFIX owl: <http://www.w3.org/2002/07/owl#> " +
        "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> " +
        "PREFIX xsd: <http://www.w3.org/2001/XMLSchema#> " +
        "INSERT DATA {" +
        "    :LaVita rdf:type owl:NamedIndividual , " +
        "            :Dipinto ; " +
        "    :haTitolo \"La Vita\"^^xsd:string . " +
        "    :PabloPicasso :creatoreDi :LaVita .\n" +
        "}");
    conn.queryResultSet(query, ResultSetFormatter.out);
}
}
```

8 Conclusions

This document presented and described the final versions of APIs that have been developed for use within the SPICE project architecture. For each API we have detailed the specifications along with its intended purpose in relation to SPICE work packages and any relevant design methodology employed.

Whilst the report inevitably includes a large overlap with deliverable D6.4, which described the SPICE project APIs in their interim state, this final version includes the development work that was carried out in M25-M36 of the project.

The report includes a user guide for each API, describing how pilot application developers can make use of the API's functions and how to make use of and customise API parameters where appropriate. Example API requests and code snippets are also supplied. Links are also made available, where appropriate, for downloadable release versions of the various software packages developed within this work package.