



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

D3.1 Prototype User and Community Model

Deliverable information	
WP	WP3
Document dissemination level	PU Public
Deliverable type	DEM Demonstrator, pilot, prototype
Lead beneficiary	UH
Contributors	UCM
Date	30/04/2021
Document status	Final
Document version	1.0

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 of 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	CELI	CELI SRL	Italy
13	UH	UNIVERSITY OF HAIFA	Israel
14	CNR	CONSIGLIO NAZIONALE DELLE RICERCHE	Italy

Executive summary

D 3.1 provides a report on the development and testing of a (interim) user and community prototype modelling in SPICE. To date, after studying the initial user modelling requirements of the different case studies, initial user model and community model data structures were defined and implemented, as well as APIs for accessing the data. A mechanism for explicitly (manually) setting and defining user modelling data (user characteristics) was developed and demonstrated to case studies users. An initial mechanism for community modelling was implemented and experimented in simulation. The main challenge we faced in this stage was lack of knowledge and uncertainty about specific requirements of case studies and the unstable situations in the museum case studies due to the COVID-19 issues that could eventually lead to some readjustments of the original case studies. This led us to develop flexible mechanisms that may accommodate diverse requirements. The details of the technology are described in this deliverable. In the next stage the specific internal reasoning mechanisms of the user and community modelling will be further developed and adapted according to emerging requirements of the case studies.

Document History

Version	Release date	Summary of changes	Author(s) -Institution
V0.1	02/04/2021	First draft released, WP3 review	Belen Diaz Agudo (UCM), Tsvi Kuflik and Alan Wecker (UH)
V0.2	21/04/2021	Internal Review	Antonio Lieto(UT) Enrico Daga (OU)
V0.3	23/04/2021	Version released to all partners for final integrations Internal review	Alan Wecker (UH)
V0.4	26/04/2021	Version for UNIBO	Alan Wecker (UH)
V1.0	30/04/2021	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 User Model	8
2.1 Motivation and justification	8
2.2 General Structure	8
2.3 Accessing the User Model	11
2.3.1 Example of Use	12
3 Community Model	13
3.1 Motivation and justification	13
3.2 Types of communities	15
3.3 Community model representation	15
3.4 Community model API	16
4 Interaction within Work Package 3	19
5 Interaction with other Work Package	21
6 Conclusions and future work	23
7 Instructions (locations of material)	24
8 References	24
8.1 User Model	24
8.2 Community Model	25
9 Appendix	26
9.1 User Model File Structure	26
9.2 SPICE-UserModel-API REST	27
9.2.1 ConfigController	27
9.2.2 PropertyController	31
9.2.3 UserController	35
9.2.4 Models	38
9.2.4.1 Configuration	38
9.2.4.2 Property	38
9.2.4.3 User	39
9.3 Screenshots	39
9.4 React example of wrapped REST calls	41
9.4.1.1 Config Service	41

9.4.1.2	User Service	42
9.4.1.3	Property Service	42

1 Introduction

WP3 Delivery 3.1 focusses on providing the technological infrastructure that enables reasoning about individuals, their characteristics and preferences, the explicit communities they identify themselves with and the implicit communities that evolve from analysing content contributed by these individuals. WP3, itself, is composed of 4 closely related components: (1) individual and community models, which are the data structures that contain information about individuals and communities (included in D3.1); (2) user and community modelling components which are the reasoning mechanisms that monitor the user continuously, reason about their behaviour and infer their preferences and community relatedness and update the models accordingly (included in D3.1); (3) textual content analysis (D3.2) that provides input data for the user and community modelling components to reason about; and (4) a recommender system (D3.4) that uses the user models and **scripts** (guidelines/instructions for activities, generated by WP6) for guiding the process of content recommendation to users.

D3.1 focusses on the individual and community user models and user modelling components. D3.1.1 documents the prototypes developed so far (first year deliverable of interim models). The document is organized as follows:

It starts with an abstract description of the user model and the user modeller and its services, and then describes the community model and community modeller and its services. Following is a description of the internal and external links of WP3. Finally, there are conclusions reached so far. An appendix includes detailed APIs, usage examples and code of the different services.

2 User Model

2.1 Motivation and justification

In the SPICE project, **user models** represent the individuals that are interacting with the system. They are key elements (together with the community models) used to guide the process of content recommendations to individuals, taking into consideration individual and community interests, as well as script guidelines (WP6), to search and identify relevant users' contributions, to provide alternative interpretations of objects, to promote the social contagion among users and to emphasize the similarities and differences within and across communities.

In this document we review the initial studies carried out to identify what user characteristics are relevant (and needed to be represented) for the project. The information gathered through a series of project meetings guided the development of the user model (a data structure containing users' information) as well as a communication protocol to access the data and a user modelling component to maintain it. The procedure was as follows: first the case studies submitted scenarios, these were examined for user and community characteristics which were categorized within known generic user model categories. The results of these items were then discussed in meetings with each of the case studies.

With respect to the user model, deliverable D3.1.1 describes the interim model for SPICE. It is the result of ongoing interactions with case studies leaders and other work package leaders.

2.2 General Structure

The deliverable consists of prototype code of REST APIs¹, built using the SPRING boot framework² and this document which describes the role which the user model plays in the overall project. In terms of deployment in the project the intention is that each case study would instantiate its own version of the REST server. In addition, a REACT frontend³ was developed which gives an example of how to use the REST services.

¹ https://en.wikipedia.org/wiki/Representational_state_transfer

² <https://spring.io/projects/spring-boot>

³ <https://reactjs.org/>

The User Model (UM) component provides three types of major services (controllers). The first concerns configuring the user model. The second concerns the creation, retrieval, update, deletion (CRUD) of users within the model. The third concerns the properties of an individual user and provides CRUD services for each property. Each case study can configure the user model to use the properties it feels necessary for its scenario. The values of the user model can be grouped to form communities implicitly in the community model.

There are a number of basic concepts:

User – This is an individual who is modelled based on **properties** that are organized into different **categories** (Identity, Demographics, Traits, Beliefs/Values, Interests, Skills, Communities, Current Contexts). A property is configured by giving it a name, how it is constrained (what the allowable values are) and an aggregation strategy (how we handle multiple calls to configure this property). Aggregation strategies can be: latest (last one given), average (mean of all values given) or weighted average (mean with later entries given more weight). In addition, when a property is added one can add information such as in what **context** the information was added, what the **source** of the information was and whether it was **explicitly** given by the user or **implicitly** derived based on some observed behaviour or other factors.

Here is further explanation of the categories and their possible use in the case study scenarios:

Identity – These are properties which identify the user (e.g., ID, email, password). Necessary if we want to use the models over more than one session.

Demographics – Descriptive of the user which are fairly stable (e.g., age, gender, place of birth). These can be used to help form explicit communities (see below)

Traits – Values which describe the user (e.g., personality, learning style). These can be used as shortcuts to determining properties.

Beliefs/Values - Items the user holds (thinks) are important/to be of value. These can be useful in the formation of implicit groups and/or common ground.

Interests - Items that the user likes. These could be evidenced by how long s/he views an artefact connected to an interest. These can be used to improve user satisfaction or find common ground

Preferences - A series of items where the user prefers A to B (triangle inequality may or may not hold i.e. $A > B, B > C, \text{ implies } A > C$). *After consultation with the case studies, we decided not to use this item.*

Skills – Things that the user is good at or believes s/he is good at. Useful for determining what scripts to run

Communities - Either explicit communities obtained from the user info or implicitly derived from community model. Used by recommender for choosing content

Current Contexts – Info about the user's current environment (system, display capabilities, weather). May be useful in determining which scripts to run.

The following table shows the categories and some of their characteristics (Structure is what items make up the values and always include the source (derivation) and date added/last updated). Item	Stability	Examples	Structure +(derivation, date)	Derivation mode (derived from)	Scenario

Identity	High	id#, password	type, name, value,	Explicit	All
Demographics	Medium high	A18Y, religion, ethnicity communities-explicit	type, name, parameter,	Explicit	DMH-Age, Gender, Education, Languages, Organizations
Traits		Personality, Learning Style, Preferred Curation Type, Current Falk Identity	type, name, degree, parameter	Explicit	
Beliefs (Values)	Medium High		type, name, degree, parameter	Derived	Hecht - Patriotic, Religious
Interests		Abstract concepts,	type, name, value on scale, {concept, activity}	Explicit (questionnaire) Implicit (User Activity)	All, implicit groups based on interests
Preferences(?)		Abstract concepts, Media	type, name1, name2, value	Explicit, or from User Activity	
Skills		Curation, Writing(Language) Reading(Language)			DMH Activities
Communities				Implicit	All
Current Context	Low	Social, Spatial, Temporal, Emotional, Environment, System			Useful for scripts

2.3 Accessing the User Model

The following is a list of the REST APIs based on the three major services described above.

The config controller is used to setup the configuration of the user model per each individual case study to allow flexibility in the design. These are basic CRUD (Create, Retrieve, Update, Delete) functionality plus an additional retrieve all function.

config-controller

PUT	/api/v2/configUpdate/{pname}	Update a property configurations by property name	↩
POST	/api/v2/configCreate	Create a property configuration for this usermodel	↩
GET	/api/v2/config	Get all configurations for this usermodel	↩
GET	/api/v2/configGet/{pname}	Get a property configurations by property name	↩
DELETE	/api/v2/configDelete/{pname}	Delete a property configurations by property name	↩

The user controller allows for additions of users to the model. Again basic CRUD functionality, with an additional helper function which lets you get a certain property across all users

user-controller

PUT	/api/v2/users2Update/{userid}	Update a user by username	↩
POST	/api/v2/users2Create	Create a new user Id and pwd should be anonymized	↩
GET	/api/v2/users2	Get all users, sorted by name	↩
GET	/api/v2/users2Get/{userid}	Get all the users properties by user name	↩
DELETE	/api/v2/users2Delete/{userid}	Delete a users by user name	↩

The property controller allows you to manage (CRUD) each specific property for each individual user. The additional functionality includes retrieving all properties, all properties of a certain user, and all properties of a certain name.

property-controller



PUT	/api/v2/propertyUpdate/{userid}	Update a specific property for a specific user	↩
POST	/api/v2/propertyCreate/{userid}	Add a new property for a user	↩
GET	/api/v2/property	Get all properties for all users	↩
GET	/api/v2/propertyGetAllByUserid/{userid}	Get all properties for a specific user	↩
GET	/api/v2/propertyGetAllByPname/{pname}	Get all properties with a certain property name	↩
GET	/api/v2/propertyGet/{userid}/{pname}	Get a specific property for a specific user	↩
DELETE	/api/v2/propertyDelete/{userid}/{pname}	Delete a specific property for a specific user	↩

See Appendix A for the details of the layout of the file structure of the code. A more detailed description of the APIs, organized by the 3 major services, can be found in Appendix B.

2.3.1 Example of Use

Screenshots of the React frontend can be found in Appendix C.

Since React doesn't know how to make REST calls directly we use the axios⁴ library to wrap the calls. In Appendix D we show the 3 services that are implemented to cover the User Model API (examples of the service calls from React to the REST APIs) and used to implement the screens in React (presented in Appendix C).

⁴ <https://github.com/axios/axios>

3 Community Model

3.1 Motivation and justification

In the SPICE project, **communities** are key elements to search and browse contents of interests, to identify similarities and differences across users and their contributions, to provide alternative interpretations of objects, to promote the social contagion among users and to emphasize the similarities and differences within and across communities.

In this document we review the initial investigations performed in SPICE to identify what types of communities are required, how to detect communities and how to represent them. We do not describe here the details of community detection and visualization, as these are planned for year 2.

Next, we review the most relevant aspects regarding the community model features, via question-answer pairs.

- **What are communities?** They are groups of users with shared characteristics. Community detection algorithms are based on similarity and can be configured to assure that each community is meaningful. All the citizens in the same community share certain characteristics. The set of characteristics depend on the data set, as a community identifies a group of users that are heavily connected among themselves, but sparsely connected to the rest. The set of characteristics to consider can be explicitly established in the **explicit** communities, but they will be detected by the clustering and community detection algorithms for the so-called **implicit** communities.
- **What features or characteristics are considered to detect a community?** Features from the user model, like similar personal profile features (demographics, age, sex) as well as features related to user opinions on items of the *content model* including the information from the artworks. A community model uses characteristics from the user model (T3.1), interactions with artworks (interpretations) (T3.2) and the content model from ontologies and linked data (WP6 and WP4).
- **What are communities for?** They support the exploration of interpretations and museum objects and help the recommender to find contents of interest. The visualization and explanation of communities allow the exploration, reflective reasoning and social cohesion of the users. Besides, communities of users that have previously used the system avoid the cold start problem as, due to intra-community similarity, a new user can be treated like other users from the same community.
- **What is a community model?** It represents the set of all the communities inferred by the community detection algorithms, their descriptions and relations. The community model can be queried using the API described below.

<https://app.swaggerhub.com/apis-docs/gjimenezUCM/SPICE-CommunityModelAPI/1.0.0>

It includes endpoints with services for communities, users and contributions and explanations.

- **May one user belong to many different communities?** Yes, different communities are formed using different sets of features, so depending on the set, the same user can be classified within different criteria. Besides, the communities depend on the whole set of users and they may change over time.
- **When does the system detect new communities?** This should be configurable. In the current version we (ideally) assume that the community model is always up-to-date. That is, each time a new user and/or a new contribution is included in the system, the community model has resources to recompute the whole community set.
- **What are the community detection algorithms used?** We are exploring the behaviour of different state of the art clustering algorithms based on similarity (k-means) and based on graph analysis (Louvain method, modularity, k-cliques, Markov Clustering). Community detection algorithms are configurable for each case study. More specifically, each use case could have a different configuration of the similarity metrics from a set of predefined options, or deciding, for example, about using (or not) overlapping communities.

- **Where is the community model stored?** It is stored in the Linked data hub infrastructure developed in WP4 which connects cultural objects, collections and citizen contributions.

3.2 Types of communities

The set of communities is dynamic and vary over time. We have already emphasized that users can belong to different communities representing different perspectives (similarity metrics) of the same user.

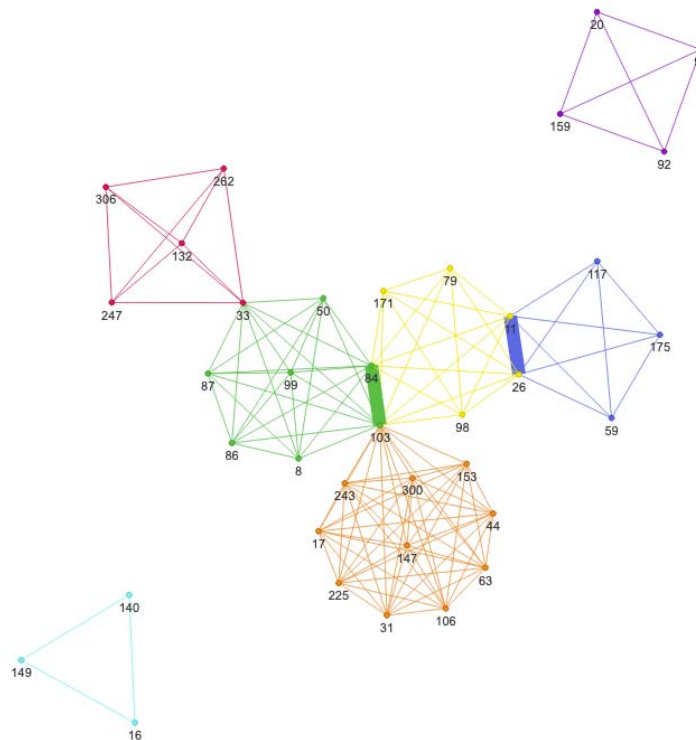
Communities can be **explicitly** stated (according to the information provided by the user model) or **implicitly** appear, i.e., *computed* by the community detection algorithms using similarity metrics. Examples of explicit communities are the personas defined in some of the project use cases (in WP7) that represent user archetypes summarizing common behaviours (like teachers in the children school visits). Other examples of explicitly stated communities could be a children group from a certain school, or visitors from an association.

Communities can also be classified as **persistent**, which are those that are stable in time and can be defined as part of the user model; or **virtual communities**, which have a temporary and dynamic character and arise with new users, new opinions, stories and/or reflections. Virtual communities are detected using the community detection algorithms and can become persistent if required.

3.3 Community model representation

Besides state-of-the-art clustering algorithms, we are also studying the use of Graph analysis algorithms to detect communities. In the later, the community model can be represented as a similarity graph (network) where nodes represent users and links represent similarity connections between them. A community identifies a group of nodes that are heavily connected among themselves, but sparsely connected to the rest of the graph. The next figure shows an example, where communities are visualized through colors. Visually, a thick line represents more similarity between the nodes.

More generally, the community model is multi-layer. Every layer uses the same set of nodes (users) and each layer represents the relationship among users using a different similarity metric. On each layer, different communities can be inferred using a similarity metric and we can find relationships between communities in different layers according to the users who belong to them. Each similarity metric could combine features (and weights for them) from the user model (T3.1), the interactions with artworks (interpretations) (T3.2) and content model (ontologies from WP4 and WP6).



We will explore community detection algorithms based on clustering (K-means, hierarchical clustering, Markov Clustering) [Xu2015] and on graph analysis (Louvain method, modularity, k-cliques) [Fortunato2010, Yang2016]. What is relevant for this task is that community detection algorithms will heavily rely on the definition of the similarity metrics. For example, we may be interested in communities of users that share similar personal profile features (demographics, age, sex) from the user model and could also combine these by identifying similar interpretations on similar contents. The use of different similarity metrics will change the set of communities in the community model and this is why we need to include configuration capabilities using a catalog of semantic similarity metrics.

Another interesting aspect to explore in the next future is the capability of explanation of the community model. That means that each community should be explained to understand, for example, why a certain user belongs to a certain community, what are the commonalities shared by the users of this community and what are the differences within other nearby communities. A community can be explained through the common shared characteristics, that are based on the similarity metric used to detect it. A symbolic description can be also built using graph-based analysis techniques (like Formal Concept Analysis) [Jorro et al 2020].

3.4 Community model API

The input of the community model are user contributions, as the interactions between a user and an artwork, according to the concepts evoked by the artwork (e.g., emotions, values).

```

contribution v {
  user*          string($uri)
                  Url to the user model information stored on linked datahub

  artwork*       string($uri)
                  Url to the artwork content information stored on linked datahub

  extended-contribution string($url)
                  example: https://...
                  Url to complete information stored about the contribution (such as timestamp, context, textual contribution, multimedia...)

  concepts
    v [
      A list of the concepts extracted from the user contribution and employed by the community model for the community detection algorithms
      string($uri)
      example: List [ "https://w3id.org/spice/SON/PlutchikEmotion/Amazement" ]
      uri to a formal concept in SPICE ontologies
    ]
}

```

The output of the Community Model (CM) is two-fold:

On one hand, it provides information about the communities inferred by the CM. This information includes a set with the specific contributions that intervened in the definition of that community. On the other hand, CM computes similarity values among the different communities.


```
community ∨ {
  id*      id string($uri)
           example: d290f1ee-6c54-4b01-90e6-d701748f0851
           Unique id

  name*    string
           example: elderly
           Community name

  explanation string
           example: People whose age is above 65
           community description (maybe empty). It can be computed by the explanation module

  community-type* string
           example: explicit
           Type of community. Virtual communities are computed by the community model. Explicit communities are provided by the user model

  size*    number
           example: 5
           Enum:
           > Array [ 2 ]
           Number of users who belong to the community

  contributions > [...]
}

```

similarityScore ∨ [

Schema for results that compute similarity values

```
similarityScore ∨ {
  targetCommunityId id string($uri)
                    example: d290f1ee-6c54-4b01-90e6-d701748f0851
                    Unique id

  otherCommunityId id string($uri)
                   example: d290f1ee-6c54-4b01-90e6-d701748f0851
                   Unique id

  value number
          Similarity value between the communities in the similarity score object

  similarity-function string
                       similarity function employed to compute this similarity score

}]

```

The community model API concerns:

- Two entry points regarding USERS:
 - One for querying about the communities that a user belongs to (GET Method)
 - One for injecting user contributions in the CM (POST Method)

users Operations related to users in communities

GET `/users/{userId}/communities` Communities that a user belongs

POST `/users/{userId}/contribution` Add user contribution

- Three entry points to query information about communities:
 - The communities within the CM (`/communities`)
 - The information about a concrete community (`/communities/{communityid}`)
 - The users that belong to a community (`/communities/{communityid}/users`)

communities Operations related to information about communities

GET `/communities` Communities in the model

GET `/communities/{communityId}` Information about a community

GET `/communities/{communityId}/users` Users who belong to a community

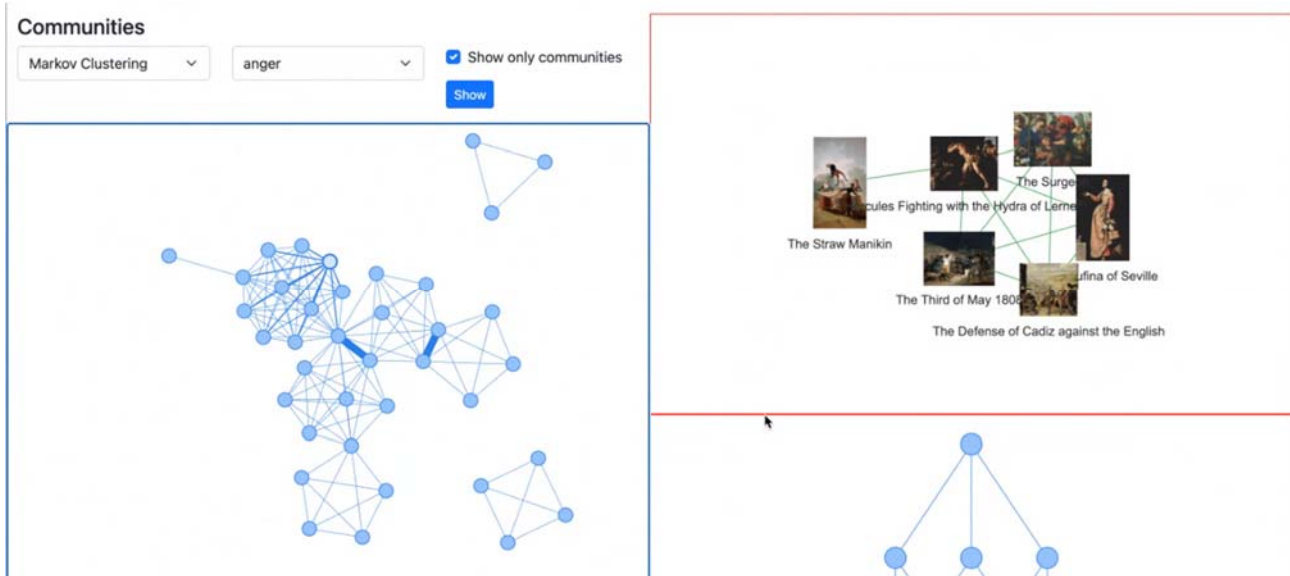
- Two entry points to provide services about **similarity between communities**:
 - A service to provide the k-most similar communities to a given one (`/communities/{communityId}/similarity`).
 - A service to compute the similarity between two given communities (`/communities/{communityId}/similarity/{otherCommunityId}`).

similarity Operations about computing similarity among communities

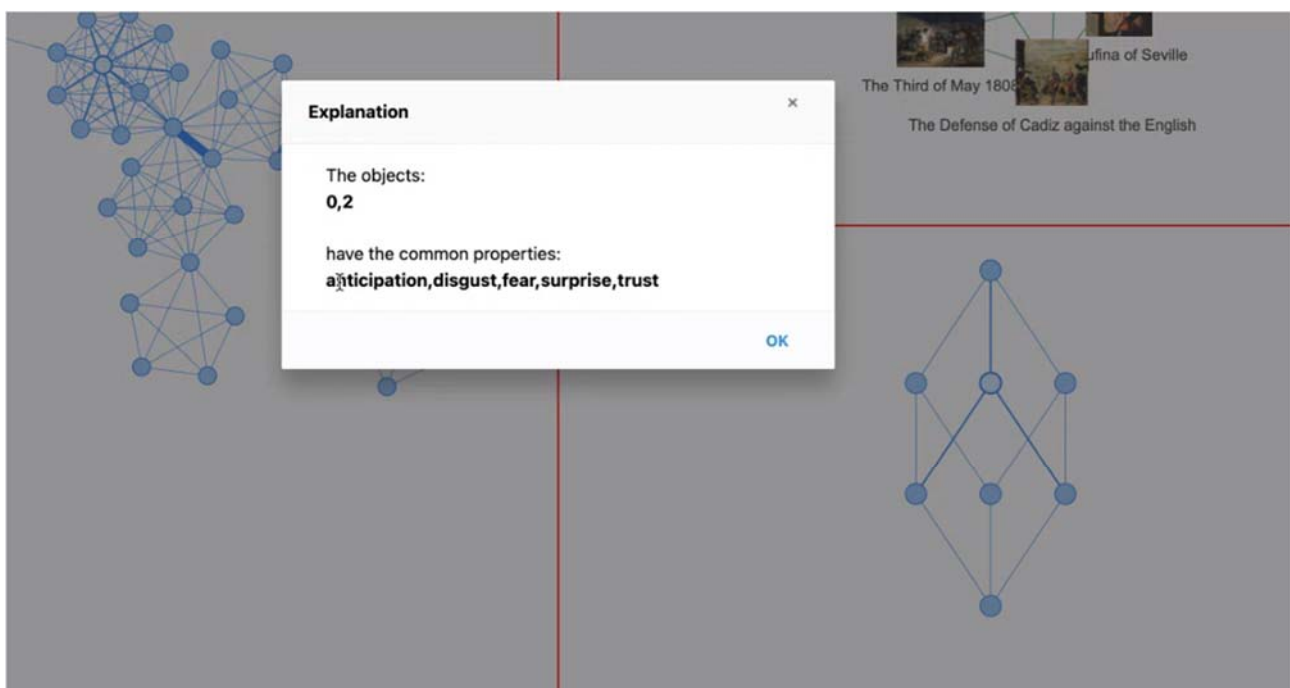
GET `/communities/{communityId}/similarity` K-most similar communities

GET `/communities/{communityId}/similarity/{otherCommunityId}` Similarity between two communities

The CM API has been tested using a preliminary prototype of a visualization tool using an example dataset of the Prado Museum, based on Wikiart Emotions dataset [Saif2018] and enhanced with synthetic data. Communities in the right can be interactively explored. In the example, two artworks are linked if they evoke the same emotion (according to Plutchik's wheel of emotions) for the users of the selected community. For each community we visualize the artworks that are representative of this community in the right-side graph.



This prototype also contains an interactive explanation (based on FCA lattices) for a given community:



4 Interaction within Work Package 3

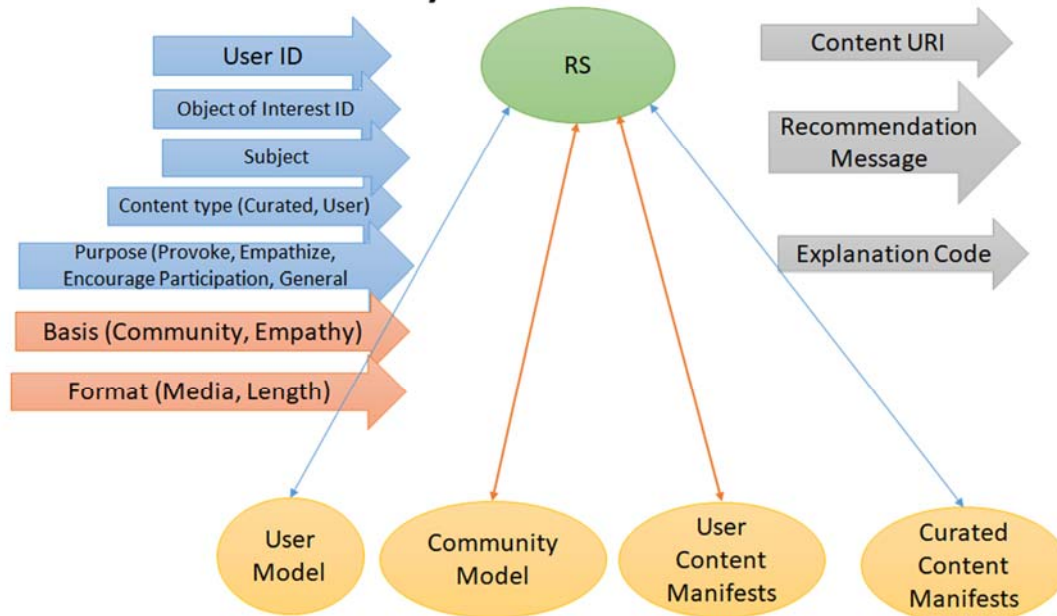
Within T3.1, bi-weekly meetings were held to co-ordinate between the user model and the community model. In addition, weekly meetings were held to monitor progress of T3.1.

With T3.2, meetings held to coordinate that the output of T3.2 can be used by the User model (and from there by the community model).

With T3.3, there was a meeting but it is less critical as these tasks are not linked. By definition, there was coordination between the community model and T3.3 as they are the same development team.

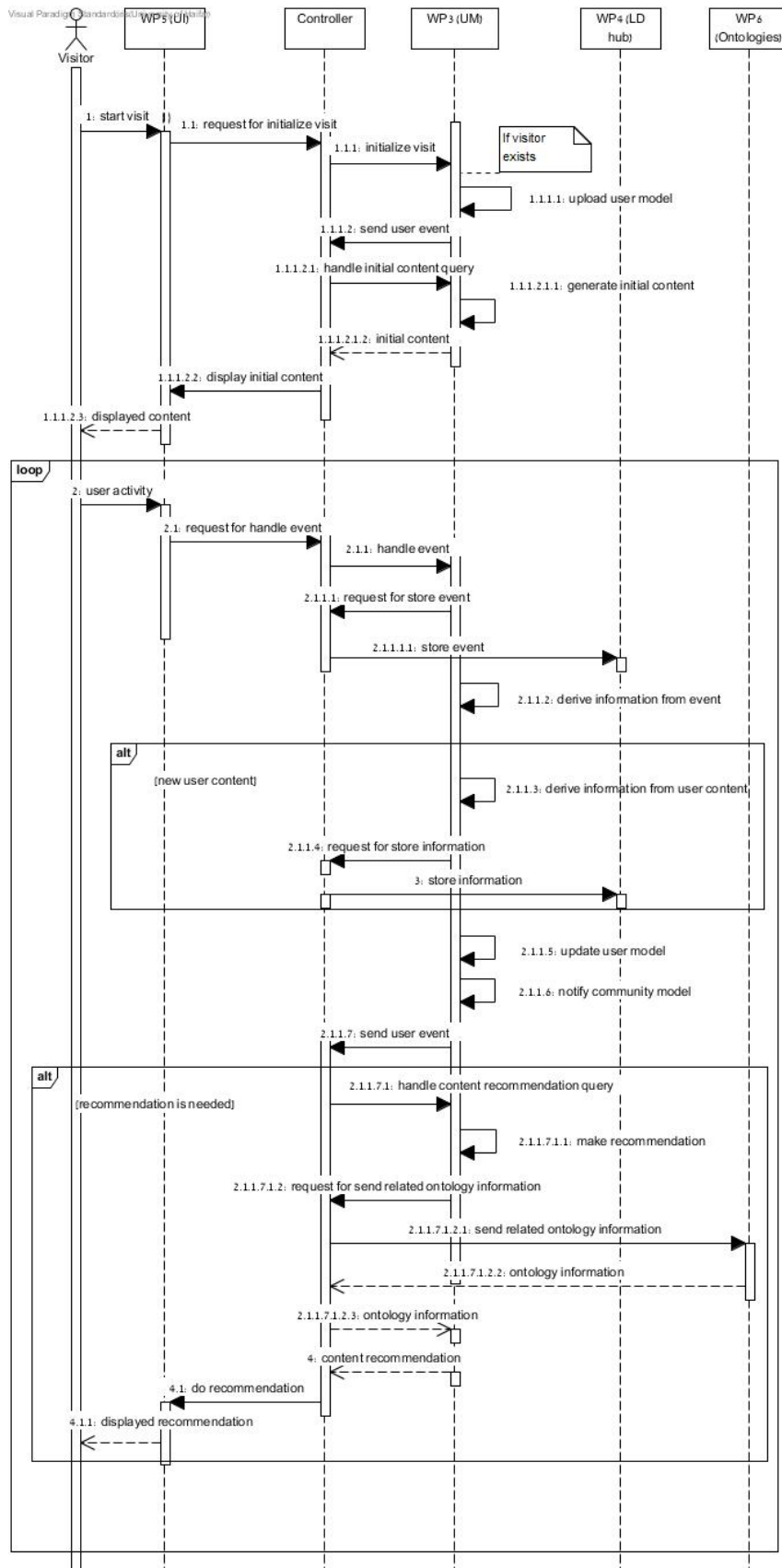
T3.4 began to examine how it could use the user and community models to provide recommendations based on this delivery.

Recommender System



5 Interaction with other Work Package

Here is diagram of the interactions of WP3 with the other packages:



With WP2 we coordinated those methods designed will provide info to the User Model and use the recommendations.

With WP4 we coordinated storage of user and community models in the Link Data Hub.

With WP5 a meeting was held to see how user and community models can be reflected in the user interface.

With WP6 a series of meetings held to coordinate that WP3 and WP6 use the same vocabularies and that the ontology reflects the structure of the user and community models.

With WP7 meetings were held with each of the case studies (IMMA, GAM, Madrid, Hecht, DMH) to understand their needs and use of the user and community models and implications for the planned recommender.

Case Study	User Characteristics	Community Characteristics
IMMA	Demographics – Children, Adults Family, Older	Based on Artwork Interpretation
GAM	Level of Physical Challenges, similar responses	Level of Physical Challenges, similar responses
Madrid	Attitudes toward climate change, Age, Rural vs Urban	Attitudes toward climate change, Age, Rural vs Urban
DMH	Age, Gender, A18Y, Occupation, Socio-Economic Status, Location, Interests, Connection to Helsinki, Education Language	City vs Non City Dwellers (Need to see scenario of asylum seekers)
Hecht	Religion, Nationality, Religiosity, Values	Religion, Nationality, Religiosity, Values

6 Conclusions and future work

In general, for task 3.1 the goals set for the first year were achieved. The heterogeneity of the case studies (which is a good thing) posed a challenge in terms of user modelling, posed a challenge on the development of the user and the community models and required us to suggest creative solutions to the uncertainty and lack of information about the intended use of the models in the case studies, which will be useful in ensuring that the models will be flexible and applicable to a wide range of scenarios. In both the user model and the community model components, continuous interaction with the case studies, flexible solutions to accommodate for changing requirements and simulations were adopted in order to allow us to achieve the first-year goals.

Regarding the User Model, the major challenge encountered (which was expected) was uncertainty in what user characteristics may be needed for modelling users, what may be explicitly provided and what will have to be inferred. The solution was a definition of a flexible user model that may be able to accommodate any user characteristic in the form of attribute-value pair, allowing the system administrator to define the relevant user model for a specific system using a dedicated configuration tool. It also allows a combination of explicit definition of user characteristics, together with an inference mechanism that is based on concepts extracted from the user generated content and the sentiment towards them, extracted by T3.2.

Future tasks include:

- Immediate- Getting the REST code to work with the JSON-LD Hub, adding aggregation methods
- Develop the reasoning mechanism for updating user preferences
- Long term – Helping the case studies utilize the user/community model
- Integrating the user model and user modelling component with the recommender system
- Integrate the user modelling mechanism into the HECHT museum case study (WP 7.3) as a prototype example for the other case studies

Regarding the Community Model, we have accomplished the goals associated with community representation and modelling including the study of services to communicate with other modules of the project. Until now, we have worked with synthetic user data and tested and reviewed some of the state of art clustering and community detection algorithms. Besides, we have started the tasks related with the development of tools for exploring aggregations of interpretations, visualization and interaction. We tested different clustering techniques for identifying commonalities and variabilities among the communities using artificial users and content. We explored with different types of communities: explicit, implicit, persistent, virtual, temporal. Some of the use cases have already reported the explicit communities they envision in their museums.

In cooperation with WP6 and WP4 we have analysed the relationships among clusters in the community model and the content concept ontologies. This initial exploration has resulted in two lines of cooperation: the first one is the use of ontologies as knowledge to compute similarity metrics that are needed in the community detection algorithms. The second line is the relation of communities with the defined concepts from the conceptual ontologies in the project. Explanation of communities will allow to identify what are the most representative terms that will be used to link the communities with the concept ontology. This will enable users to browse the ontologies and the repository of content through the community model. This cooperation needs to be extended in the future to define semantic similarity based on ontological content, and allowing some of the implicit communities identified through the algorithms to be included as concepts in the ontologies (only for stable communities).

Various clustering techniques, like K-means or formal concept analysis have already been experimented although we need to explore further to identify the most effective techniques for the task. The homogeneous groups of similar interpretations will be used to identify and represent the "interpretation archetypes" that will support the design of recommendation models tailored for the different user communities developed as part of task 3.1.

We now summarize future work related with the Community Model:

- Extend the literature study regarding clustering techniques and community modelling. We will include all the literature study in the prototype of clustering techniques (Deliverable 3.5, month 24).
- Experiment with different clustering techniques for identifying commonalities and variabilities among the communities. Extended examples with real data will be developed.
- Extend the initial literature study of similarity and dissimilarity for intra-community and inter-community. Examples and report are expected to be included in Deliverable 3.5.
- Analyze further relationships among detected communities in the CM and high-level concepts in the concept ontology.
- Develop an exploration tool to browse the repository of content and the communities through the vocabulary of the ontology, generate visual explanations from the relation intra communities and inter communities and generate higher-level ontology concepts from semantic annotation of clusters.

7 Instructions (locations of material)

This document can be found at ZENODO (DOI according to version) 10.5281/zenodo.4708753

The source code Version 1,1 for the User Model can be found at ZENODO 10.5281/zenodo.4724887. Latest version is 10.5281/zenodo.4724886.

A draft version of the user model REST API can also be found at:

<https://app.swaggerhub.com/apis/ajwecker/SPICE-UserModel-API/v0#/user-controller>

A draft version of the community model REST API is available at: <https://app.swaggerhub.com/apis-docs/gjimenezUCM/SPICE-CommunityModelAPI/v1>

8 References

8.1 User Model

Alexandridis, G., Chrysanthi, A., Tsekouras, G. E., & Caridakis, G. (2019). Personalized and content adaptive cultural heritage path recommendation: an application to the Gournia and Çatalhöyük archaeological sites. *User Modeling and User-Adapted Interaction*, 29(1), 201–238. <https://doi.org/10.1007/s11257-019-09227-6>

Antoniou, A., Katifori, A., Roussou, M., Vayanou, M., Karvounis, M., Kyriakidi, M., & Pujol-Tost, L. (2016). *Capturing the Visitor Profile for a Personalized Mobile Museum Experience: an Indirect Approach*. <http://chess.madgik.di.uoa.gr:10005/cvs->

Cena, F., Likavec, S., & Rapp, A. (2019). Real World User Model: Evolution of User Modeling Triggered by Advances in Wearable and Ubiquitous Computing. *Information Systems Frontiers*, 21(5), 1085–1110. <https://doi.org/10.1007/s10796-017-9818-3>

Leung, R., & Law, R. (2010). A review of personality research in the tourism and hospitality context. *Journal of Travel and Tourism Marketing*, 27(5), 439–459. <https://doi.org/10.1080/10548408.2010.499058>

Musto, C., Semeraro, G., Lovascio, C., De Gemmis, M., & Lops, P. (2018). A framework for holistic user modeling merging heterogeneous digital footprints. *UMAP 2018 - Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization*, 97–101. <https://doi.org/10.1145/3213586.3226218>

Roussou, M., & Katifori, A. (2018). Flow, staging, wayfinding, personalization: Evaluating user experience with mobile museum narratives. *Multimodal Technologies and Interaction*, 2(2). <https://doi.org/10.3390/mti2020032>

Spallazzo, D. (2012). *Sociality and Meaning Making in Cultural Heritage Field. Designing the Mobile Experience*.

8.2 Community Model

Fortunato, S. Community detection in graphs, *Physics Reports*, vol 486, Issues 3–5, (2010) 75-174, ISSN 0370-1573, <https://doi.org/10.1016/j.physrep.2009.11.002>.

Jose Luis Jorro, Marta Caro-Martínez, Belén Díaz-Agudo, Juan A. Recio-García: A User-Centric Evaluation to Generate Case-Based Explanations Using Formal Concept Analysis. *ICCBR 2020*: 195-210 978-3-030-58341-5 https://doi.org/10.1007/978-3-030-58342-2_13

Saif M. Mohammad and Svetlana Kiritchenko. WikiArt Emotions: An Annotated Dataset of Emotions Evoked by Art. In *Proceedings of the 11th Edition of the Language Resources and Evaluation Conference (LREC-2018)*. (2018), Miyazaki, Japan.

Yang, Z., Algesheimer, R. & Tessone, C. A Comparative Analysis of Community Detection Algorithms on Artificial Networks. *Sci Rep* 6, 30750 (2016). <https://doi.org/10.1038/srep30750>

Xu, D., Tian, Y. A Comprehensive Survey of Clustering Algorithms. *Ann. Data. Sci.* 2, 165–193 (2015). <https://doi.org/10.1007/s40745-015-0040-1>

Frank Y. Guo, Sanjay Shamdasani, Bruce Randall, Creating Effective Personas for Product Design: Insights from a Case Study. *International Conference on Internationalization, Design and Global Development. IDGD 2011: Internationalization, Design and Global Development* pp 37-46| *Lecture Notes in Computer Science* book series (LNCS, volume 6775)

9 Appendix

9.1 User Model File Structure

The structure of the file system is:

```
/usermodel
/usermodel/src/main/java
    il.ac.haifa.is.spice
    il.ac.haifa.is.spice.controller
    il.ac.haifa.is.spice.exception
    il.ac.haifa.is.spice.model
    il.ac.haifa.is.spice.repository
    il.ac.haifa.is.spice.security
/usermodel/src/main/resources
    /usermodel/src/main/resources/application.properties
/usermodel/src/test/java
/usermodel/doc
/usermodel/react-frontend (example frontend)
    /usermodel/react-frontend/build
    /usermodel/react-frontend/node_modules
    /usermodel/react-frontend/public
    /usermodel/react-frontend/src
    /usermodel/react-frontend/src/components
    /usermodel/react-frontend/src/services
    /usermodel/react-frontend/src/App.css
    /usermodel/react-frontend/src/App.js
    /usermodel/react-frontend/src/App.test.js
    /usermodel/react-frontend/src/index.css
    /usermodel/react-frontend/src/index.js
    /usermodel/react-frontend/src/logo.svg
    /usermodel/react-frontend/src/reportWebVitals.js
    /usermodel/react-frontend/src/setupTests.js
    /usermodel/react-frontend/debug.log
    /usermodel/react-frontend/package-lock.json
    /usermodel/react-frontend/package.json
    /usermodel/react-frontend/README.md
/usermodel/src
/usermodel/target
/usermodel/HELP.md
/usermodel/mvnw
/usermodel/mvnw.cmd
/usermodel/pom.xml
/usermodel/usermodel-api-docs.json
```

9.2 SPICE-UserModel-API REST

A more detailed description of the APIs again organized by the 3 major services

9.2.1 ConfigController

POST /api/v2/configCreate

Create a property configuration for this usermodel (createConfiguration)

Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

Request body

body [Configuration](#) (required)

Body Parameter —

Return type

[Configuration](#)

Example data

Content-Type: application/json

```
{
  "aggregationStrategy" : "Latest",
  "pname" : "pname",
  "ptype" : "ptype",
  "constraint" : "constraint",
  "id" : 0,
  "category" : "Identity"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK [Configuration](#)

DELETE /api/v2/configDelete/{pname}

Delete a specific property configuration by property name (deleteConfiguration)

Path parameters

pname (required)

Path Parameter —

Return type

map[String, Boolean]

Example data

Content-Type: application/json

```
{
  "key" : true
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK

GET /api/v2/config

Get all configurations for this usermodel (getAllConfigurations)

Return type

array[[Configuration](#)]

Example data

Content-Type: application/json

```
[ {
  "aggregationStrategy" : "Latest",
  "pname" : "pname",
  "ptype" : "ptype",
  "constraint" : "constraint",
  "id" : 0,
  "category" : "Identity"
}, {
  "aggregationStrategy" : "Latest",
  "pname" : "pname",
  "ptype" : "ptype",
  "constraint" : "constraint",
  "id" : 0,
  "category" : "Identity"
} ]
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK

GET /api/v2/configGet/{pname}

Get a property configurations by property name (getConfigurationByPname)

Path parameters

pname (required)

Path Parameter —

Return type

[Configuration](#)

Example data

Content-Type: application/json

```
{
  "aggregationStrategy" : "Latest",
  "pname" : "pname",
  "ptype" : "ptype",
  "constraint" : "constraint",
  "id" : 0,
  "category" : "Identity"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK [Configuration](#)

PUT /api/v2/configUpdate/{pname}

Update a property configuration by property name (updateConfiguration)

Path parameters

pname (required)

Path Parameter —

Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

Request body

body [Configuration](#) (required)

Body Parameter —

Return type

[Configuration](#)

Example data

Content-Type: application/json

```
{
  "aggregationStrategy" : "Latest",
  "pname" : "pname",
  "ptype" : "ptype",
  "constraint" : "constraint",
  "id" : 0,
  "category" : "Identity"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK [Configuration](#)

9.2.2 PropertyController

POST /api/v2/propertyCreate/{userid}

Add a new property for a user (createProperty)

Path parameters

userid (required)

Path Parameter —

Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

Request body

body [Property](#) (required)

Body Parameter —

Return type

[Property](#)

Example data

Content-Type: application/json

```
{
  "pname" : "pname",
  "context" : "context",
  "pvalue" : "pvalue",
  "id" : 6,
  "source" : "source",
  "userid" : "userid"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK [Property](#)

DELETE /api/v2/propertyDelete/{userid}/{pname}

Delete a specific property for a specific user (deleteUser1)

Path parameters

userid (required)

Path Parameter —

pname (required)

Path Parameter —

Return type

map[String, Boolean]

Example data

Content-Type: application/json

```
{
  "key" : true
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK

GET /api/v2/propertyGetAllByPname/{pname}

Get all properties with a certain property name (getAllPropertyByPname)

Path parameters

pname (required)

Path Parameter —

Return type

array[[Property](#)]

Example data

Content-Type: application/json

```
[ {
  "pname" : "pname",
  "context" : "context",
  "pvalue" : "pvalue",
  "id" : 6,
  "source" : "source",
  "userid" : "userid"
}, {
  "pname" : "pname",
  "context" : "context",
  "pvalue" : "pvalue",
  "id" : 6,
  "source" : "source",
  "userid" : "userid"
} ]
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK

GET /api/v2/propertyGetAllByUserId/{userid}

Get all properties for a specific user (getAllPropertyByUserId)

Path parameters

userid (required)

Path Parameter —

Return type

array[[Property](#)]

Example data

Content-Type: application/json

```
[ {
  "pname" : "pname",
  "context" : "context",
  "pvalue" : "pvalue",
  "id" : 6,
  "source" : "source",
  "userid" : "userid"
}, {
  "pname" : "pname",
  "context" : "context",
  "pvalue" : "pvalue",
  "id" : 6,
  "source" : "source",
  "userid" : "userid"
} ]
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK

GET /api/v2/property

Get all properties for all users (getAllPropertys)

Return type

array[[Property](#)]

Example data

Content-Type: application/json

```
[ {
  "pname" : "pname",
  "context" : "context",
  "pvalue" : "pvalue",
  "id" : 6,
  "source" : "source",
  "userid" : "userid"
}, {
  "pname" : "pname",
  "context" : "context",
  "pvalue" : "pvalue",
  "id" : 6,
  "source" : "source",
  "userid" : "userid"
} ]
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK

GET /api/v2/propertyGet/{userid}/{pname}

Get a specific property for a specific user (getPropertyByUserIdAndPropertyName)

Path parameters

userid (required)

Path Parameter —

pname (required)

Path Parameter —

Return type

[Property](#)

Example data

Content-Type: application/json

```
{
  "pname" : "pname",
  "context" : "context",
  "pvalue" : "pvalue",
  "id" : 6,
  "source" : "source",
  "userid" : "userid"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK [Property](#)

PUT /api/v2/propertyUpdate/{userid}

Update a specific property for a specific user (updateProperty)

Path parameters

userid (required)

Path Parameter —

Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

Request body

body [Property](#) (required)

Body Parameter —

Return type

[Property](#)

Example data

Content-Type: application/json

```
{
  "pname" : "pname",
  "context" : "context",
  "pvalue" : "pvalue",
  "id" : 6,
  "source" : "source",
  "userid" : "userid"
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK [Property](#)

9.2.3 UserController

POST /api/v2/users2Create

Create a new user Id and pwd should be anonymized (createUser)

Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

Request body

body [User](#) (required)

Body Parameter —

Return type

[User](#)

Example data

Content-Type: application/json

```
{
  "password" : "password",
  "id" : 0,
  "userid" : "userid",
  "properties" : [ {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
    "source" : "source",
    "userid" : "userid"
  }, {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
    "source" : "source",
    "userid" : "userid"
  } ]
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK [User](#)

DELETE /api/v2/users2Delete/{userid}

Delete a user by userid (deleteUser)

Path parameters

userid (required)

Path Parameter —

Return type

map[String, Boolean]

Example data

Content-Type: application/json

```
{
  "key" : true
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

/

Responses

200

OK

GET /api/v2/users2

Get all users, sorted by name (getAllUsers)

Return type

array[[User](#)]

Example data

Content-Type: application/json

```
[ {
  "password" : "password",
  "id" : 0,
  "userid" : "userid",
  "properties" : [ {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
    "source" : "source",
    "userid" : "userid"
  }, {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
    "source" : "source",
    "userid" : "userid"
  } ]
}, {
  "password" : "password",
  "id" : 0,
  "userid" : "userid",
  "properties" : [ {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
    "source" : "source",
    "userid" : "userid"
  }, {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
    "source" : "source",
    "userid" : "userid"
  } ]
} ]
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

• */*

Responses

200

OK

GET /api/v2/users2Get/{userid}

Get all the users properties by user name (getUserByUserId)

Path parameters

userid (required)

Path Parameter —

Return type

[User](#)

Example data

Content-Type: application/json

```
{
  "password" : "password",
  "id" : 0,
  "userid" : "userid",
  "properties" : [ {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
    "source" : "source",
    "userid" : "userid"
  }, {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
    "source" : "source",
    "userid" : "userid"
  } ]
}
```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK [User](#)

PUT /api/v2/users2Update/{userid}

Update a user by username (updateUser)

Path parameters

userid (required)

Path Parameter —

Consumes

This API call consumes the following media types via the Content-Type request header:

- application/json

Request body

body [User](#) (required)

Body Parameter —

Return type

[User](#)

Example data

Content-Type: application/json

```
{
  "password" : "password",
  "id" : 0,
  "userid" : "userid",
  "properties" : [ {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
```

```

    "source" : "source",
    "userid" : "userid"
  }, {
    "pname" : "pname",
    "context" : "context",
    "pvalue" : "pvalue",
    "id" : 6,
    "source" : "source",
    "userid" : "userid"
  } ]
}

```

Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- */*

Responses

200

OK [User](#)

9.2.4 Models

This is a list of the different objects (Configuration, Property, User) used by the API

9.2.4.1 Configuration

id (optional)

[Long](#) format: int64

category (optional)

[String](#)

Enum:

Identity

Demographics

Traits

Beliefs

Interests

Skills

Communities

CurrentContexts

pname (optional)

[String](#)

pvalue (optional)

[String](#)

constraint (optional)

[String](#)

aggregationStrategy (optional)

[String](#)

Enum:

Latest

Average

Decay

Weighted

9.2.4.2 Property

id (optional)

[Long](#) format: int64

userid (optional)

[String](#)

pname (optional)

[String](#)

pvalue (optional)

[String](#)

source (optional)

[String](#)

context (optional)

[String](#)

9.2.4.3 User

id (optional)

[Long](#) format: int64

userid (optional)

[String](#)

password (optional)

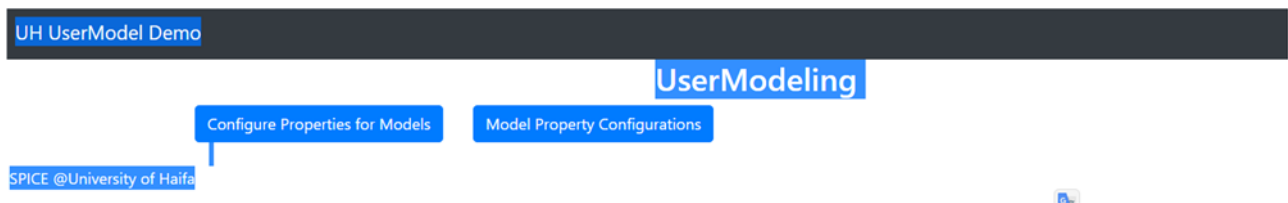
[String](#)

properties (optional)

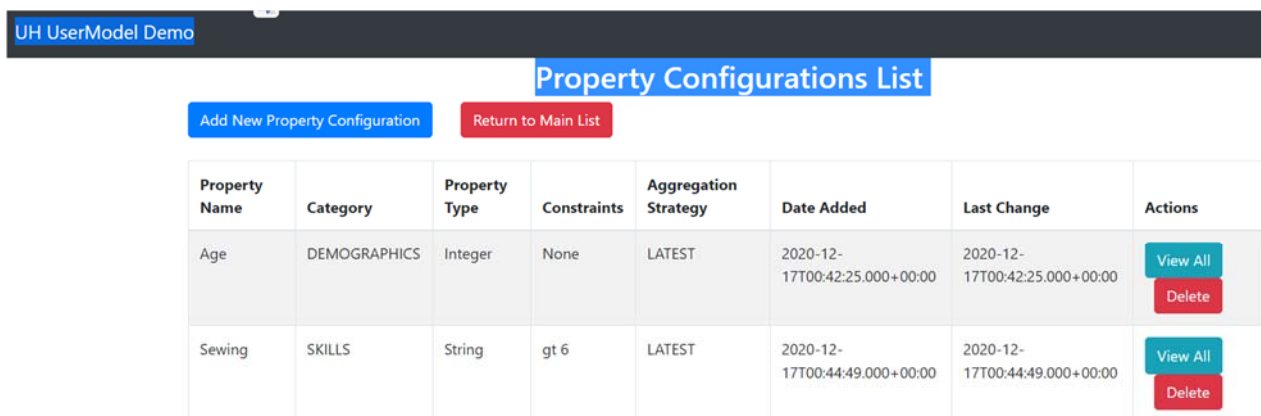
[array/Property](#)

9.3 Screenshots

Initial screen gives choice to either (1) configure a User Model or (2) work with the configured model



List of different properties in the configurations (From choice 1 in First screen)



Property Name	Category	Property Type	Constraints	Aggregation Strategy	Date Added	Last Change	Actions
Age	DEMOGRAPHICS	Integer	None	LATEST	2020-12-17T00:42:25.000+00:00	2020-12-17T00:42:25.000+00:00	View All Delete
Sewing	SKILLS	String	gt 6	LATEST	2020-12-17T00:44:49.000+00:00	2020-12-17T00:44:49.000+00:00	View All Delete

SPICE @University of Haifa

An example of all user properties with name of Age from previous screen View All

UH UserModel Demo

Property (Age) List

[Return to Main List](#)

User	Property Name	Property Value	Source	Context	Timestamp	Actions
MDAwMDAw	Age	53	explicit	questionnaire	2020-12-17T02:42:45.000+00:00	
ABqMQj31	Age	54	explicit	questionnaire	2020-12-17T10:45:41.000+00:00	

SPICE @University of Haifa

List of all users from first screen option 2

UH UserModel Demo

User List

[Add New User](#)
[Return to Main Screen](#)

Name Anonymized	User Password	Date Added	Last Change	Actions
ABqMQj31	italyski	2020-12-10T08:49:40.000+00:00	2020-12-10T08:49:40.000+00:00	View Delete
bUtBcYL7UI	pw3	2020-12-13T22:09:17.000+00:00	2020-12-13T22:09:17.000+00:00	View Delete
MDAwMDAwMD	pw1	2020-12-09T04:02:28.000+00:00	2020-12-10T02:21:28.000+00:00	View Delete

SPICE @University of Haifa

View values for a particular user from previous screen

UH UserModel Demo

User (MDAwMDAwMD) Properties List

[Add Property](#)
[Return to User List](#)

User	Property Name	Property Value	Source	Context	Timestamp	Actions
MDAwMDAw	Age	53	explicit	questionnaire	2020-12-17T02:42:45.000+00:00	Update Delete
MDAwMDAw	Extrovert	Medium	explicit	questionnaire	2020-12-14T16:13:59.000+00:00	Update Delete
MDAwMDAw	birthplace	Tel Aviv	explicit	questionnaire	2020-12-14T16:13:25.000+00:00	Update Delete

SPICE @University of Haifa

9.4 React example of wrapped REST calls

9.4.1.1 Config Service

```
import axios from 'axios';
const USER_API_BASE_URL = "http://localhost:8080/api/v2/config";
//
class ConfigService {
  getConfigurations(){
    return axios.get(USER_API_BASE_URL);
  }
  createPropertyConfiguration(config){
    return axios.post(USER_API_BASE_URL+"Create", config);
  }
  getConfigByName(propertyName){
    return axios.get(USER_API_BASE_URL + 'Get/' + propertyName);
  }
}
```

```

    updateConfig(config, propertyName){
        return axios.put(USER_API_BASE_URL + 'Update/' + propertyName, config);
    }
    deleteConfig(propertyName){
        return axios.delete(USER_API_BASE_URL + 'Delete/' + propertyName);
    }
}
export default new ConfigService()

```

9.4.1.2 User Service

```

import axios from 'axios';
const USER_API_BASE_URL = "http://localhost:8080/api/v2/users2";
//
class UserService {
    getUsers(){
        return axios.get(USER_API_BASE_URL);
    }
    createUser(user){
        return axios.post(USER_API_BASE_URL+"Create", user);
    }
    getUserById2(userId){
        return axios.get(USER_API_BASE_URL + 'Get/' + userId);
    }
    updateUser2(user, userId){
        return axios.put(USER_API_BASE_URL + 'Update/' + userId, user);
    }
    deleteUser2(userId){
        return axios.delete(USER_API_BASE_URL + 'Delete/' + userId);
    }
    getPropertyByUserIdPropertyName(propertyName, userId){
        return axios.get(USER_API_BASE_URL + '/' + userId+'/'+propertyName);
    }
}
export default new UserService()

```

9.4.1.3 Property Service

```

import axios from 'axios';
const PROPERTY_API_BASE_URL = "http://localhost:8080/api/v2/property";
class PropertyService {
    getPropertyByUserId(userId){
        return axios.get(PROPERTY_API_BASE_URL+'GetAllByUserId/'+userId);
    }
    getPropertyByName(pname){
        return axios.get(PROPERTY_API_BASE_URL+'GetAllByPname/'+pname);
    }
    createProperty(property, userId){
        return axios.post(PROPERTY_API_BASE_URL+'Create/'+userId, property);
    }
}

```

```
getPropertyById(propertyName, userid){
    return axios.get(PROPERTY_API_BASE_URL+'Get/' + userid+'/'+propertyName);
}
updateProperty(property, userid){
    return axios.put(PROPERTY_API_BASE_URL + 'Update/' + userid, property);
}

deleteProperty(userid, propertyName){
    return axios.delete(PROPERTY_API_BASE_URL+'Delete/' + userid+'/'+propertyName);
}
}
export default new PropertyService()
```