DELIVERABLE REPORT D5.1 "Content Abstraction and Databases"

collaborative project

MASELTOV
Mobile Assistance for Social Inclusion and Empowerment of Immigrants with Persuasive Learning Technologies and Social Network Services

Grant Agreement No. 288587 / ICT for Inclusion

project co-funded by the
European Commission
Information Society and Media Directorate-General
Information and Communication Technologies
Seventh Framework Programme (2007-2013)

Due date of deliverable:	31 December, 2013 (month 24)
Actual submission date:	07 August, 2014 (Revision 1)
Start date of project:	Jan 1, 2012
Duration:	36 months

Work package	**WP5 – PERSONALIZATION AND RECOMMENDATION**
Task	**T5.1 – Content Abstraction and Databases**
Lead contractor for this deliverable	**AIT**
Editor	**Sofoklis Efremidis**
Authors	**Sofoklis Efremidis, Iakovos Georgiou**
Quality reviewer	**Mical Busta, CTU**

Project co-funded by the European Commission within the Seventh Framework Programme (2007–2013)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

partner		organisation	ctry
01	JOANNEUM RESEARCH DIGITAL	JOANNEUM RESEARCH FORSCHUNGSGESELLSCHAFT MBH	AT
02	cure	CURE – CENTER FOR USABILITY RESEARCH AND ENGINEERING	AT
03	ATHENS INFORMATION TECHNOLOGY	RESEARCH AND EDUCATION LABORATORY IN INFORMATION TECHNOLOGIES	EL
04	Universitat Oberta de Catalunya IN3	UNDACIO PER A LA UNIVERSITAT OBERTA DE CATALUNYA	ES
05	The Open University	THE OPEN UNIVERSITY	UK
06	Coventry University	COVENTRY UNIVERSITY	UK
07	m p	CESKE VYSOKE UCENI TECHNICKE V PRAZE	CZ
08	FH JOANNEUM HEALTH CARE ENGINEERING / eHEALTH	FH JOANNEUM GESELLSCHAFT M.B.H.	AT
09	TELECOM ITALIA	TELECOM ITALIA S.p.A	IT
10	Fluidtime Design Software Service	FLUIDTIME DATA SERVICES GMBH	AT
11	busuu.com the language learning community!	BUSUU ONLINE S.L	ES
12	Fundeso Fundación Desarrollo Sostenido COMPARTIENDO OPORTUNIDADES	FUNDACION DESARROLLO SOSTENIDO	ES
13	DANAIDA	VEREIN DANAIDA	AT
14	mrc	THE MIGRANTS' RESOURCE CENTRE	UK

CONTENT

1. EXECUTIVE SUMMARY

This document reports on the underlying concepts and abstractions for the personalized services offered by the MASELTOV platform. The personalization of the MASELTOV services depends on a set of user data and preferences and also the user context, which is captured through a set of events coming from a number of sensors and Mapp applications. The document presents how the user context, which is continuously updated as a result of user actions, is communicated from the client User Profile to the back end server for storage and subsequent manipulation. The user context, which comprises information like the current user position and movement, user activity like browsing, searching, and user interactions with other users, forms the basis for the MASELTOV personalized services and targeted recommendations. Back end components like the recommender pick on the stored contextual information for producing personalized recommendations in an attempt to enhance the overall user experience and facilitate social inclusion.

2. OVERVIEW

The rich suite of applications that are offered by the MASELTOV platform includes targeted personalized recommendation services, which help immigrant users to adapt easier to their new environment and facilitate their social inclusion. Personalized recommendations are generated based on contextual user information as well as their profile data and declared preferences. Contextual user information is formed as a result of fusing data coming from a variety of sensors on the smartphone device, like GPS receiver, accelerometer, etc. or by monitoring user behavior. Obviously the more information pertaining to a user is available the more specialized and targeted the produced recommendations can be.

Contextual information captures the environment and behavior of the user. Information related to the user environment may include the user's current position, ambient weather conditions, user's movement, etc. Typically, this information is captured by a variety of sensors on board the user's smartphone and relayed to the User Profile for further storage and processing. Moreover, user behavior relates to user actions, like the use of a MApp application, the searching of the wiki for a specific term, the searching of the internet for a topic of interest, or a place of interest, etc. As a result of such actions, a number of events are generated by the various MApp applications, which are also sent to the User Profile for storage and subsequent processing.

Raw data coming from the user environment or the user actions are processed for extracting user related information and fused for the formulation of abstractions about the user context. Context abstractions can subsequently form the basis for targeted and personalized recommendations for the user, which may further be enhanced by the user preferences as specified in the User Profile structures.

This deliverable presents the architecture of the MASELTOV platform components that are responsible for processing and storing of user contextual information, the way this information is structured and stored in the back end databases, and its subsequent processing towards the generation of personalized recommendations.

3. USER PROFILE AND RECOMMENDER

The User Profile is a central component of the MASELTOV platform as it maintains structures like user data and preferences and is also the sink for events and notifications that are generated by Mapp applications and smartphone services, respectively. All this information is eventually relayed and stored in a back end database, the structure of which is shown in the following sections. The stored information captures the user context which forms the basis for subsequent processing, and the generation of personalized recommendations to the user. The whole process takes place in real time, i.e., as contextual information is continuously updated as a result of user actions, it is directly used by the recommender to produce recommendations to the user. The overall architecture of the User Profile and recommender components is shown in Figure 1. As shown in the figure, both components comprise a client and a server subcomponent. The back end database is the repository for the user related data and preferences and the events and notifications coming from the client side. These data are used by the recommender for generating recommendations to the user. The recommender also makes use of a set of rules, which reside is a special purpose file that resides outside the back end database.

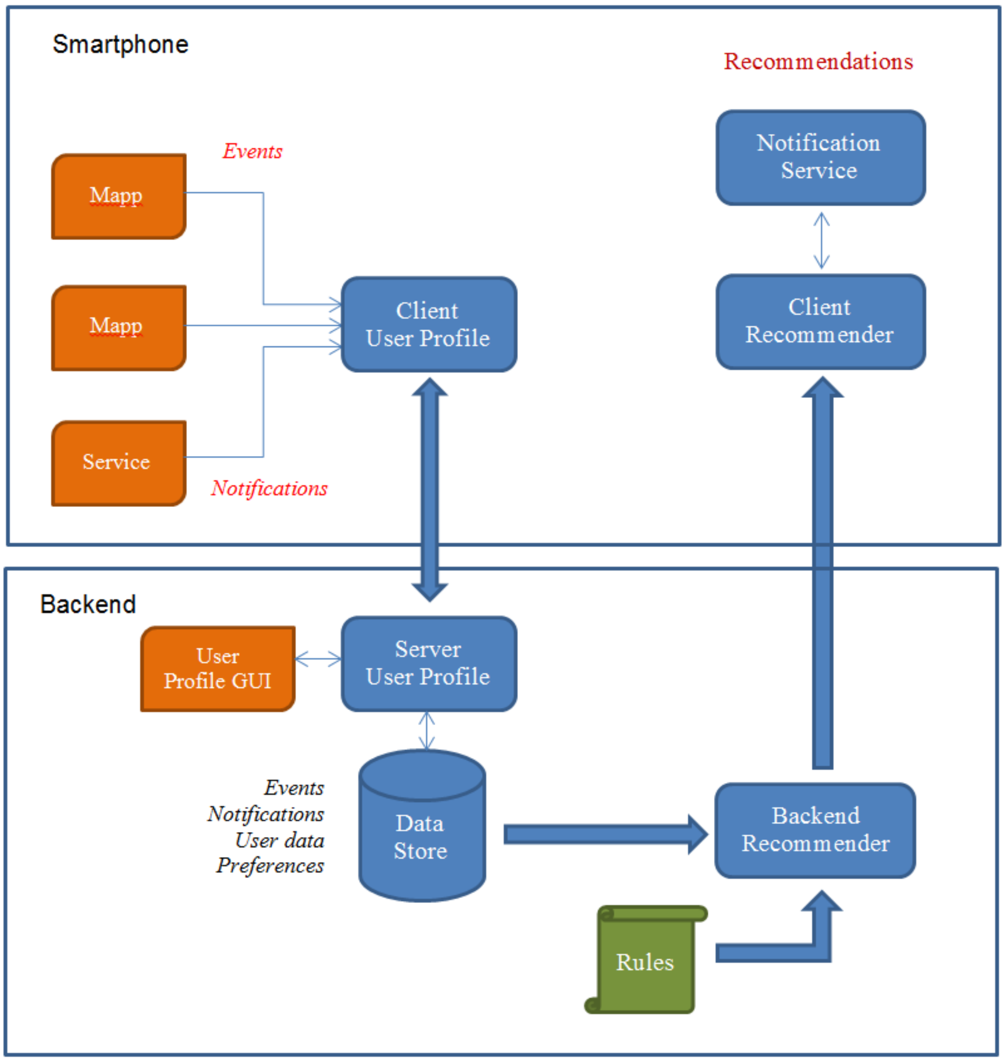


Figure 1: User Profile and Recommender Architecture.

4. CONTENT ABSTRACTIONS

User content includes any information that pertains to user preferences and user actions. User preferences are a set of static data that are declared by the user and reflect his/her likes, e.g. hobbies, cultural, entertainment, recreational preferences and so on. User actions pertain to user activity and state, like the current user location and movement, interactions with other users, use of tools and applications and so on. User actions are captured by events that are continuously generated and relayed to the User Profile for storage and subsequent processing.

Events and notifications are generated by a number of Mapp applications and services and carry data related to the user context. Events are produced by a number of diverse Mapp applications and services, for example the location service that makes use of the onboard GPS sensor, the TextLens application that makes user of the onboard camera, the language learning Mapp application, the social radar, etc. Each Mapp application generates and sends to the User Profile events according to the user actions, for example starting a new topic for a language course will generate an event. The generated events continuously capture the user context, they are relayed to the back end server, where they are stored so as to be used by other services like the recommender.

The diversity of the generated events hints towards a generic definition of their structure so that they can carry information from a multitude of sources and in a uniform way. Therefore the definition of events as presented also in Deliverable D5.2 "User Profiling and Personalization" should be as generic as possible and the information they carry should be expressed in a symbolic form. The structure of events as defined in Deliverable D5.2 contains the following fields.

1. its source identification, i.e., a unique id of the component that has produced the event
2. a timestamp, indicating the time the event has been produced, and
3. a collection of <key, value> pairs that specify the information that is carried along with the event.

Item (3) of the list above is purposely defined in such a generic way. Different applications generate different types of contextual data, which must be carried through in a uniform way. More concretely an event has the following structure (expressed in Java notation).

```
String source;
GregorianCalendar timestamp;
HashMap<String, Object> info;
```

The structure above is directly mapped to an Android *ContentValues* object, which contains all the fields above, and can be communicated to the User Profile Content Provider through an insert URI, which is also specified in D5.2 "User Profiling and Personalization". The same structure is used for notifications coming from Mapp services, for example, the positional service. Subsequent sections show how the generic event structure that is presented above is eventually mapped to database tables.

Each Mapp application has its own unique source string, for example, events generated by the personalized learning application have as value of the source field "Learning". When the mixed reality game reaches a point where a point of interest is shown in the background of a scene, it will send an event that has the value “MixedRealityGame” for the source field and {“ImmigrationOfficeBuilding”, <location>)} for the info field.

The combination and fusion of raw data that are carried by a number of Mapp events results into the extraction of information and knowledge about the user context and the subsequent formulation of content abstractions. The extracted information can be further used for advanced services, for example, the building of a dynamic user profile, the production of personalized recommendations, etc.

The MASELTOV platform specifies a recommender component, whose purpose is exactly this, i.e., to generate targeted recommendations from the abstracted user content. The recommender is driven by a set of rules, which specify what is to be recommended based on the detected abstract user content. In its most general form, a rule specifies what action to be taken (recommendations in the case of MASELTOV) when certain conditions are satisfied. The general form of a rule is

Predicate → Action [expiration specification]?

The interpretation of this rule is that when the predicate is satisfied the rule fires and the action is taken. Therefore, in the case of the Recommender rules, the predicate captures the user content as abstracted from data that are carried through events, and the action is the recommendation to be issued based on the abstracted content.

An example rule is the following:

Predicate:
dist(current location, 37o58’51 12”N, 23o45’15 81”E) < 1Km
&&
User has recently searched for keyword “Music”
Action:
Recommend attending tonight’s concert at Athens Music Hall

Informally, the rule says that if the user is currently located within a 1 kilometer radius from the Athens Music Hall and has recently launched a search related to music a recommendation should be issued for an upcoming music event in the Music Hall.

The abstracted user content comprises two parts: the current context, which includes user location, and the recent user behaviour. The rule will fire if the predicate, which expresses the abstracted user content becomes true. In turn, the user content is abstracted from the events that are relayed to the User Profile and are logged in the back end database.

Deliverable D5.4.1 contains a formal specification of rules and in particular of their preconditions. For the sake of completeness the grammar specification of rule preconditions is shown below as well.

```
pred : | aexpr

aexpr : bterm { ‘||’ bterm }*

bterm : bfactor { ‘&&’ bfactor }*
```

```
bfactor : cfactor
          [[ '==' | '!=' | '>' | '>=' | '<' | '<=' ] cfactor ]?

cfactor : dfactor { [ '+' | '-' ] dfactor }*

dfactor : efactor  { [ '*' | '/' | '%' ] efactor }*

efactor : [ '!' | '-' ] efactor
        | UserProfEntry
        | EventEntry
        | predef '(' expr-list ')'
        | number
        | 'true'
        | 'false'
        | string
        | '[' lat ',' long ']'
        | '(' aexpr ')'

expr-list :
          | expr { ',' expr }*

EventEntry: '$' event-name '.' event-field
```

The above grammar specifies the form of rule preconditions. References to events are specified as well as shown by the EventEntry resolution, which is used in the efactor rule. Therefore references to events and the carried <key, value> pairs are allowed by the grammar. The same grammar allows the use of a set of predefined predicates, like *dist*, which is shown in the example above. An actual implementation may provide support for a number of predefined predicates. For example, the current prototype implements the following primitives:

- findInfoComponentResource() - returns the resource matching related keyword of the Info article's title (used in “Info Article Title Recommendation” rule).
- findTextLensComponentResource() - returns the resource matching related keyword of the translated text (used in “TextLens Recommendation” rule).
- finLanguageLearningLModuleByPOI() - returns the appropriate language learning module based on the category of the POI that the user has entered; the module is going to be used to send the user directly to the specific Language Learning module (used in “User Enters a POI Recommendation” rule).
- checkUserPreferencesWithPOIs(POIs) - returns a number of POIs (in JSON format) that match their categories with one of the user's preference hobbies (used in “User Preferences and POIs” rule).
- userHasHobbies() - returns a boolean true, if the user has already at least one hobbies selected in user profile, otherwise returns false (used in “User Preferences Hobbies is Empty” rule).
- setLearningLevel(language, course, level) - sets a new level for a language/course combination for the user (used in “Learning Level Recommendation” rule).
- getUserCoins() - returns the number of coins currently available for the user (used in “Add Coins Recommendation” rule).

A detailed list of the implemented rules that are referenced in the previous lists as well as the corresponding recommendations is given in Deliverable 5.4.2 “Recommendation Services”.

5. DATABASES

This section shows the structure of the back end database that is used by the User Profile and the Recommender components of the MASELTOV platform. Figure 2 depicts the Entity Diagram of the database. A short description of the tables shown in the diagram is given in the sequel.

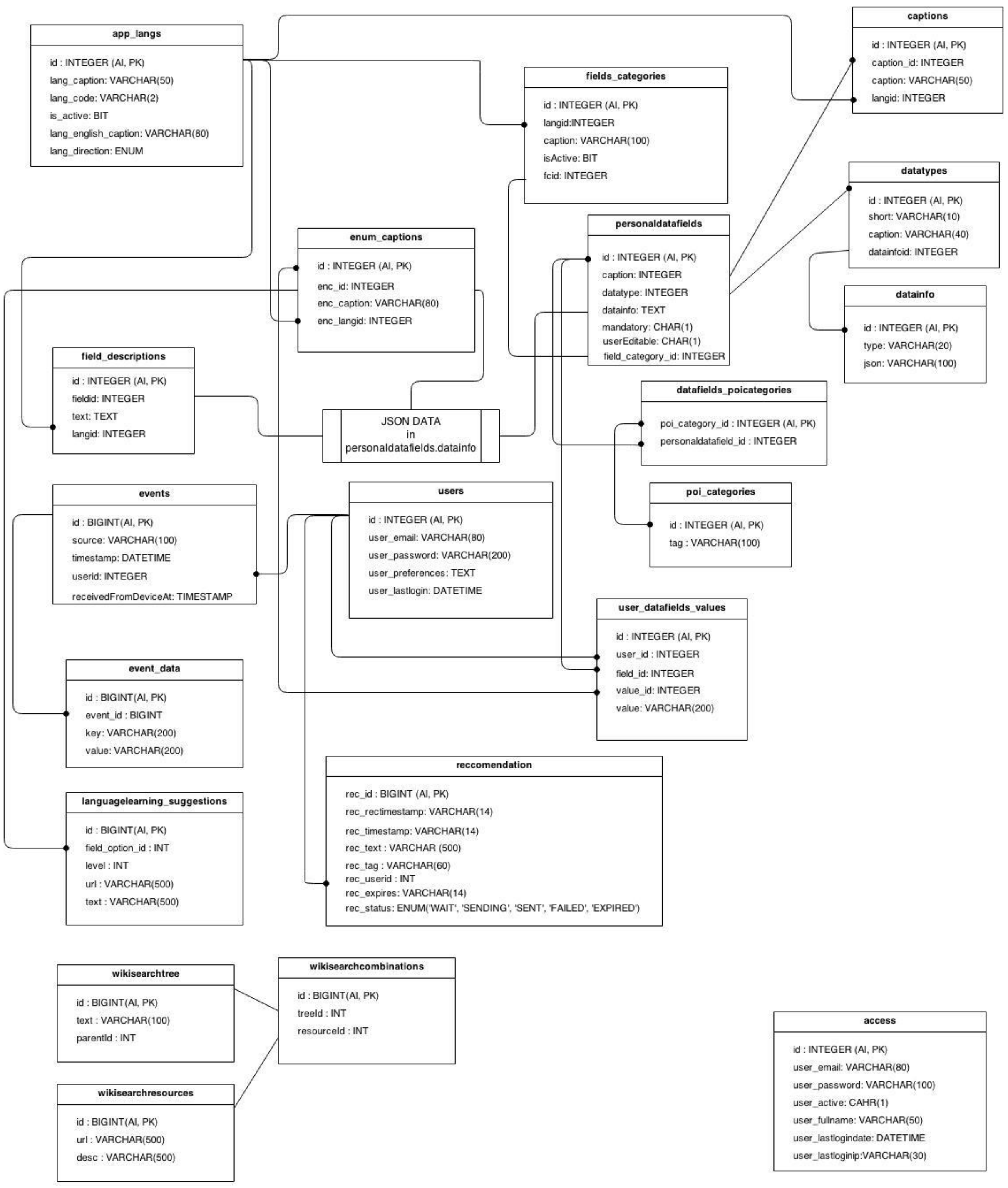


Figure 2: Database ERD Diagram.

Figure 2 is the relational diagram of the back end database. The key tables are *users* and *personadatafields*; a short explanation is given below.

- Table *users* holds information about users i.e., their username, password, last login timestamp and a JSON string (for caching purposes) of the user's preferences. *Users* table is a reference for tables *recommendations*, *events* and *user_datafields_values*.
- Table *personaldatafields* holds the list of the defined fields that store user preferences. A number of associated tables store the details and values of the preferences for the registered user (like data_types, captions, enum_caption, field_descriptions and datafields_poicategories).

Personaldatafields.datainfo field contains a JSON string that holds the user's preferences. Some of them contain a plain text value and some of them hold a reference id to the respective record of the table where the actual value resides in. These references concern the tables *field_descriptions* and *enum_captions*. So, for example the user's selection for the field gender may be 'male' or 'female'. Each one of these two options has their own id and the actual value resides in the *enum_captions* table. The later holds the option's value in all supported languages.

Table *user_datafields_values* holds a record for each user and each user's preference value. For the user's preferences that hold multiple selection values, *user_datafields_values* table includes multiple entries for the given user id and field id.

Since the application has multilingual support, tables that contain texts to be displayed to the user, need to hold all language related texts. Table *app_langs* holds the list of available languages therefore tables that contain texts (like captions, enum_captions, field_decriptions, fields_categories) have a reference to *app_langs* table.

Events received from the Mapp applications are recorded to the *events* table with reference to *users.id* field, therefore each event directly relates the user whose actions resulted to the event. Moreover, the event's specific information is recorded separately to *event_data* table (with reference to *events.id*). Events may keep coming from various sources (MApp applications and services) and for different user; once they are stored in the *events* table they can be queried by back end applications like the recommender.

Recommendations table contains the recommendations created for all users with a reference to the *users.id* field so as to maintain s reference to the user to whom it is destined.

Tables *wikisearchtree*, *wkisearchresources* and *wikisearchcombinations* are only related to each other (as shown in Figure 2) but have no other relation with the rest of the database's tables. These tables contain data that are used to identify a resource from the data coming from the events and in case there is a match, a recommendation string is created that is stored in the recommendations table. The administrator can set a number of keywords in the *wikisearchtree* table and a number of resources in *wikisearchresources* table.

Table *access* holds information about the backend user and has no other active relation with the rest of the tables.

Following is a detailed description of each database table and their fields.

Table Name	access
Description	This table keeps a record for each user of the MASELTOV administrator
Columns	id: a unique id for the rows of the table, each user has a unique id user_email: the user's email user_password: the user's password (encrypted) user_active: denotes if the user's account is active (1) or inactive (0) user_fullname: the user's fullname user_lastlogindate: the date and time of the user's last successful login user_lastloginip: the user's IP of the last successful login

Table Name	users
Description	This table keeps a record for each user of the MASELTOV application.
Columns	id: a unique id for the rows of the table, each user has a unique id user_email: the user's email user_password: the user's password (encrypted) user_preferences: holds the user's preferences in JSON, used as a caching mechanism to sever faster the API requests from the device user_lastlogin: the date and time of the last successful login (using the API from the device)

Table Name	user_datafields_values
Description	This table keeps a number of records (two or more) for each user (from the users table). Each row in this table for each user joins the field id from the personaldatafields table and the value the user has selected for this field. The value may be a string (in `value` field) of the id of the value of an enumeration type option (`value_id` field). It is actually the exploded JSON of users.user_preferences field
Columns	id: a unique id for each user's preference selection user_id: a look up to users.id; connects these values with a user field_id: a look up to personaldatafields.id; connects the value to a specific field value _id: a look up to enum_captions.id; connects the to a fields option value: the value entered by the user for this field (if not a predefined field's option)

Table Name	app_langs
Description	This table keeps a record for each language available in MASELTOV application
Columns	id: a unique id for each language supported by the application lang_caption: the language title written in the language's alphabet lang_code: the language's code (ISO 639-1) is_active: denotes if the user is active (1) or inactive (0) lang_english_caption: the language title in English lang_direction: denotes the text orientation for the language (ltr for left-to-right and rtl for right-to-left)

Table Name	personaldatafields
Description	This table keeps a record for each user preference field

Columns	id: a unique id for each field caption: the id of the field’s title looking at captions.caption_id datatype: the id of the field’s data type looking at datatypes.id datainfo: a string in JSON format, used as caching mechanism to serve faster information about the fields mandatory: denotes if the fields is mandatory (1) or not (0) userEditable: denotes if the field is editable by the user (1) or not (0)

Table Name	captions
Description	This table keeps a record of the title of each user’s preference field (from the *personaldatafields* table) for each language
Columns	id: a unique id for each caption caption_id: a look up to the personaldatafields.caption ; determines for which field the data are caption: the title of the field (at the language’s alphabet; i.e., what the user will see) langid: a look up to the app_langs.id determines the language

Table Name	datatypes
Description	This table keeps a record of the different types of data for the user’s preference fields (from the personaldatafields table)
Columns	id: a unique id for the available data types of the fields short: a short name of the data type caption: a name of the data type to display to the backend user datainfoid: a look up id to the datainfo.id; determines which the datainfo template to use

Table Name	datainfo
Description	This table keeps the template json describing each data type
Columns	id: a unique id for each data info type: the type of the field that its data info template can be applied json: the template (in JSON format) to be used for each new field of a specific data type

Table Name	field_descriptions
Description	This table keeps the titles of each user preference field (from the personaldatafields table) for each language
Columns	id: a unique id for each field’s description fieldid: a look up to the personaldatafields.datainfo JSON value for the field’s description text: a text written in the language’s alphabet, for describing why and how MASELTOV uses the user’s data langid: a lookup to the app_langs.id which distinguishes the field’s descriptions for each language

Table Name	enum_captions
Description	This table keeps all titles for each option of a user’s preference field of type enumeration, for each language
Columns	id: a unique id for each caption for each field’s options

	enc_id: a look up to the personaldatafields.datainfo JSON value for the field's option enc_caption: the title of the field's option (in the alphabet of each language) enc_langid: a lookup to app_langs.id which determines the field's option for each language

Table Name	events
Description	This table keeps a row for each event received from the MASELTOV applications from all users
Columns	id: a unique id for each event sent from the devices userid: a look up to the users.id (determines which user's device sent the event) source: the source of the event (which Mapp component produced this event) timestamp: the date and time the event was sent by the Mapp component to the client side user profile receivedFromDeviceAt: the timestamp that holds the moment that the event received by the backend server

Table Name	event_data
Description	This table keeps a number of records (one or more) for each event (from the event table) and contains the key and value of the information describing this event
Columns	id: a unique id for each event's data event_id: a lookup to events.id; connecting the event data information with an event key: the key of the event data as sent by the Mapp component value: the value of the event data as sent by the Mapp component

Table Name	poi_categories
Description	This table keeps a record of the tags to be recognized across the POI JSON
Columns	id: a unique id for each POI category tag: a tag that describes the POI category

Table Name	datafields_poicategories
Description	This table keeps the join information between the poi categories tags (from poi_categories table) with the options of the user's preference field of hobbies. This join result the categories of points of interest with the user's interests
Column	poi_category_id: a look up to poi_categories.id personaldatafield_id: a look up to user's preference field

Table Name	recommendation
Description	This table keeps a record of each recommendation created by the recommender system.
Columns	rec_id: a unique id for each recommendation rec_rectimestamp: the date and time the recommendation created on server rec_timestamp: the date and time the last triggered event rec_text: the text produced for this recommendation rec_tag: the tag string that was recognized in order to produce the

	recommendation (for the User.Location events) rec_userid: a look up to users.id; determines the user for which the recommendation is produced rec_expires: the date and time that the recommendation expires (NULL if never expires) rec_status: the status of the recommendation (Wait, Sending, sent, Failed, Expired)

Table Name	languagelearning_suggestions
Description	This table keeps a record of the resources to recommend to the user for each language/course and level he/she reaches.
Columns	id: a unique id for each language learning suggestion field_option_id: a look up to the field that this suggestion applies to level: the level that this suggestion refers to url: the URL that will be recommended to the user if this level is achieved text: a description for the administrator in order to distinguish the suggestions

Table Name	wikisearchtree
Description	This table keeps the keywords recognised for the wiki search. This table keeps a number of keywords in a two level tree format. The keywords are matched with the user's searches and upon a match a resource will be recommended to the user
Columns	id: a unique id for each item in the wiki search tree text: the title of the tree item parentId: the parent id of another item in this table (zero if the item has parent the root)

Table Name	wikisearchresources
Description	This table keeps a record of the resources available for the user recommendations. The administrator sets a number of resources that can be recommended to the user according to what the user was looking in the wiki component
Columns	id: a unique id for the wiki search resource url: the URL to recommend to the user for this resource desc: a description for each resource to help the administration distinguish them

Table Name	wikisearchcombinationss
Description	This table is an intermediate table that connects the keywords with the resources for the wiki search recommendations.
Columns	id: a unique id for each wiki search combination treeId: a lookup to the field wikisearchtree.id resourceId: a lookup to the field wikisearchresources.id

The actual schema of the back end database is given in Appendix A.

6. DATABASE CONTENTS

This section gives the details of the contents of the back end database, namely the way user preferences are events are stored.

6.1 USER PREFERENCES

The back end database holds basic information for each user in table *users*. The information maintained for each user includes:

- user’s unique id
- user’s e-mail
- one-way encoded password
- last login timestamp
- registration timestamp and
- a field named *user_preferences* which holds the personal preferences of the user formed as a JSON string.

The field *user_preferences* is a cached string with the intention of minimizing the response time during the user login process. As soon as the user login credentials are entered, they are sent to the backend server through the User Profile API. In case of a successful authentication the API responds with the user’s preferences (as a JSON string) so they can be used to build the User Profile structure within the client component. The API uses the *users.user_preferences* field to read the JSON string directly and send it to the mobile client without the need to construct it on the fly by making (expensive) queries and joins to the various database tables. Therefore, the JSON string that resides in the *user_preferences* field serves as a cached structure for completing the user login process faster.

On the other hand, every time the user makes changes to any User Profile fields, the changed values are used to update the *user_datafields_values* table, and reconstruct the JSON string of the *users.user_preferences* field (cache update) so that it remains fresh and ready for the next login request of the user.

In summary, the same data (user’s preferences) are maintained by the database in two different forms so as to be used efficiently by two different mechanisms:

1. The login API request. It must respond fast returning a JSON string that contains the user’s preferences; all of them should be returned with no requirement for processing any one of its fields.
2. The recommender. It needs to use some of the preferences fields for checking rule predicates, so the data (user’s preferences) must be structured so as to achieve high performance.

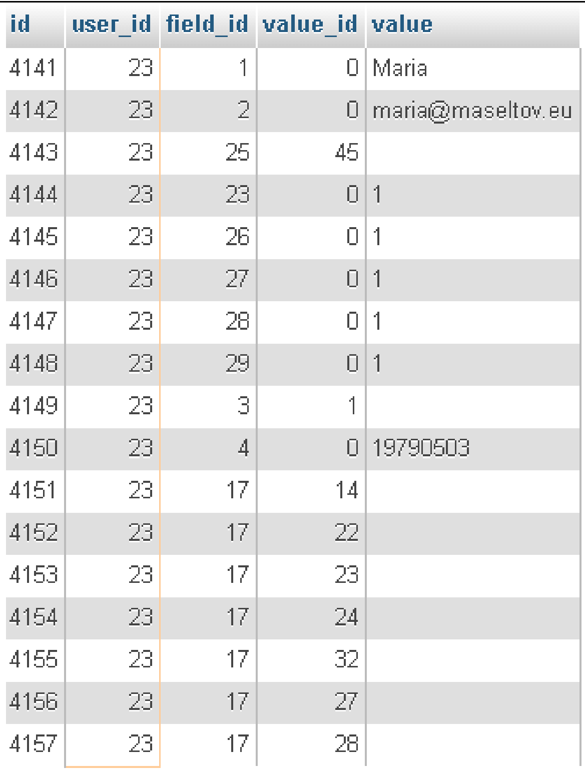
Table 1 shows two different formats of the user’s data.

Table 1: User's Preferences in JSON Format and As Separate Records.

JSON String in users.user_preference

```
{
  "fields": [
    {
      "id": "1",
      "value": "Maria"
    },
    {
      "id": "2",
      "value": "maria@maseltov.eu"
    },
    {
      "id": "25",
      "value": "{\"id\":\"45\"}"
    },
    {
      "id": "23",
      "value": "1"
    },
    {
      "id": "26",
      "value": "1"
    },
    {
      "id": "27",
      "value": "1"
    },
    {
      "id": "28",
      "value": "1"
    },
    {
      "id": "29",
      "value": "1"
    },
    {
      "id": "3",
      "value": "{\"id\":\"1\",\"value\":\"English\"}"
    },
    {
      "id": 4,
      "value": "19790503"
    },
    {
      "id": 17,
      "value":
"[{\"id\":\"14\"},{\"id\":\"22\"},{\"id\":\"23\"},{\"id\":\
"24\"},{\"id\":\"32\"},{\"id\":\"27\"},{\"id\":\"28\"}]"
    }
  ]
}
```

User_datafields_values



id	user_id	field_id	value_id	value
4141	23	1	0	Maria
4142	23	2	0	maria@maseltov.eu
4143	23	25	45	
4144	23	23	0	1
4145	23	26	0	1
4146	23	27	0	1
4147	23	28	0	1
4148	23	29	0	1
4149	23	3	1	
4150	23	4	0	19790503
4151	23	17	14	
4152	23	17	22	
4153	23	17	23	
4154	23	17	24	
4155	23	17	32	
4156	23	17	27	
4157	23	17	28	

Table user_datafields_values holds multiple records for each user (*user_id* field). Each of these records specifies the User Profile field (*field id*) and the current value for the field. User preferences that are left empty by the user do not appear in this table. The fields may hold either text type values (in this case the actual value is stored in field *users_datafields_values.value*), or may hold a predefined value (as defined by the admin, through the backend management interface, e.g., Male and Female are the two available values for field Gender) and in this case the field *user_datafields_values.value_id* is used as a reference to the field's value. Table 2 summarizes the different options for the values of the User Profile fields.

Table 2: Data stored in user_datafields_values.

Elements in fields JSON Object	**Columns in user_datafields_values**	
id	field_id	
value	**For fields of data types: Enumeration, Multi-level Boolean Enumeration, Multi-level Integer Enumeration**	**For fields of data types: Text, Number, Number (limited) E-mail, Date, Language, On/Off Switch**
	value_id	value

6.2 EVENTS

Each event is described by a set of three variables:

- *timestamp*: specifies the date and time the event is created (YYYYMMDDHHIISS format)
- *source*: specifies the event source, i.e., a unique identifier of the type of event
- *info*: is a JSONObject that contains one or more fields providing the specific information of the event

The following list gives some examples of event data.

- Event generated by GPS Tracking
```
{
        "timestamp": "20140508162004",
        "source" : "User.Source",
        "info" : {
                "longitude" : "16.346948",
                "latitude"    : "48.217689"
        }
}
```

- Event generated by TextLens
```
{
        "timestamp": "20140512152014",
        "source" : "TextLens",
        "info" : {
                "detectedText":"please take one"
        }
}
```

- Event generated by Mode of Transportation
```
{
        "timestamp": "20140512152014",
        "source" : "MaseltovContext.ModeOfTransportation",
        "info" : {
                "type":"walking",
                "confidence":"92"
        }
}
```

When an event is sent to the User Profile the id of the user from whom this event is generated is also communicated. The user id is communicated directly to the API and is not contained with the event structure. The information that is carried by an event is split into two tables by the back end User Profile components: *events* and *event_data*.

The event source, timestamp and user's id are saved into the *events* table. Since the events may be received with significant delay (for example the user's device may not be connected to the Internet when the event is created by a Mapp component) the API also records in events table the date and time (*events.receivedFromDeviceAt* field) that the event was received by the API of the User Profile client component.

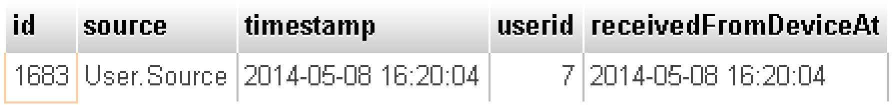
For each one of the events, back end User Profile will also create a number of records in the *event_data* table. With respect to the previous example for GPS Tracking, the User Profile will create two records in *event_data* table, one with *event_data.key*=<latitude> and one with *event_data.key*=<longitude>. Both records will have a reference to the event id (*event_data.event_id*) and will have the *event_data.value* fields filled with values 48.217689 and 16.346948, respectively.

So, the event with data

```
        {
                "timestamp": "20140508162004",
                "source" : "User.Source",
                "info" : {
                        "longitude" : "16.346948",
                        "latitude"    : "48.217689"
                    }
        }
```

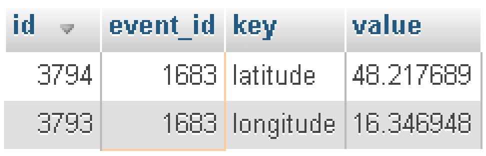
will produce the following records in the database.

event table



id	source	timestamp	userid	receivedFromDeviceAt
1683	User.Source	2014-05-08 16:20:04	7	2014-05-08 16:20:04

event_data table



id	event_id	key	value
3794	1683	latitude	48.217689
3793	1683	longitude	16.346948

6.3 USE OF EVENTS AND PREFERENCE DATA

User preferences that form the static part of the User Profile are stored in the *personaldatafields* table. Moreover, events coming from Mapp applications and services (and capture the user context as was explained above) are stored in the *events* table and are immediately available for querying and further processing. For example, back end services, like the recommender, query the *events* table when attempting to match rules for producing recommendations. Multiple such queries may run concurrently either while attempting to fire various rules, or executing on behalf of several users. The overall efficiency of executing such queries is obviously limited by the capabilities of the back end database server. Typical database servers (even freely available ones like MySQL that is used for the present prototype implementation) are multithreading and make use of connection pools to boost performance.

7. USER IDENTIFICATION

The User Profile client component communicates with the backend server through the API that is offered by the server. The API responds to certain URI requests in two ways:

1. It responds with data that are user independent (hence no authentication is required). Such requests are:
    a. List of fields (/upfields/): The returned list does not contain any values the user may have set for these fields. This API request is used in order to retrieve the list and type of fields and so as the client User Profile component to build the lists of fields.
    b. Field details (/upfield/fid): Returns information about a single field, *fid*.
    c. List of languages (/langs/): Returns the list of available languages in order to be able to build the list of languages in client User Profile Mapp so as the user to select their preferred language.
    d. List of User Profile drawer titles (/upcategories/): Returns a list of titles to be able to build the left drawer menu list in Mapp User Profile.
2. It responds with user related data where the request must be authenticated. Such requests are:
    a. Usage statistics (/usagestats/): Returns a list of Mapp components and the time the user has used the component.
    b. New event (/event/): Records a new event coming from Mapp while a user is logged in, in the application.
    c. Update user's preference (/userup/): Records the changed values of one or more user's preferences.
    d. User login (/user/): Returns a list of values for user preferences. For each field in the user preferences the API returns a field id and the user's selected value. These values are used to fill the Mapp User Preferences list with the user's selections.
    e. Creation of new user (/newuser/): Creates a new user; sets the data filled in by the user in MApp registration form.
    f. List of recommendations (/recomm/): Returns a list of recommendations produced by the Recommender.
    g. Update recommendation (/uprecomm/): Records the change of status on a specific recommendation (e.g., recommendation is starred, recommendation is read etc.).

For the URIs requests above, the MApp requests must provide e-mail and password of the logged in user. The e-mail is communicated as a two-way encrypted string, while the password is communicated as a one-way encoded string.

The following is a sample URI request for user login.

http://maseltov.ait.gr/maseltov/wservice/user/2f59747435312e44cd8ca3e1f1e7d6bd/d8578edf8458ce06fbc5bb76a58c5ca4

The format of the request is

<protocol>://<domain>/maseltov/wservice/<action>/<user_email>/<user_password>

The following is a sample URI request for a new event.

> http://maseltov.ait.gr/maseltov/wservice/event/2f59747435312e44cd8ca3e1f1e7d6bd/d8578edf8458ce06fbc5bb76a58c5ca4?timestamp=20140514130521&source=User.Location&info={“duration’:”25.12926545”}&id=120

The format of the request is

> <protocol>://<domain>/maseltov/wservice/<action>/<user_email>/<user_password>/?timestamp=<timestamp>&source=<source>&info=<event_info_json>&id=<event_id_reference>

The above sample URIs define the following API parameters.

1. <action>: the action to be taken, either *user* (for login in a user) or *event* (for communicating an event).
2. <user_email>: a two-way (AES) encrypted string that contains the user’s e-mail.
3. <user_password>: a one-way (MD5) encrypted string produced from the user’s password.

The API upon receiving such a request, it first identifies the user and then proceeds to execute the appropriate action. The API decrypts the e-mail received and seeks a match in the database (table *users*). As soon as the user exists and the password is matched, the API records the action for the specific user. In case the user validation process fails, the API responds with the following error message and won’t execute the action received.

```
{
        "result": {
        "user": {
                "id": null,
                "reason": "WRONG_CREDENTIALS"
                }
        }
}
```

MASELTOV allows users to register and then login without using their real e-mail address. Instead they can use a “fake” e-mail address produced automatically by MApp User Profile by using unique ids from the device. This allows the user to login from their device without exposing their real e-mail address. The above described communication via the MASELTOV API works exactly the same way in case of anonymous users, since the MApp User Profile creates a unique e-mail address and password for such a user. These credentials are also transmitted using the same encryption rules as a “real” user’s credentials.

The prototype implementation of the User Profile and the back end database is based on the XAMPP 1.8.2 and MySQL 5.0.10 servers. Both can handle multiple users and requests at the same time so simultaneous activities from concurrent users are handled automatically by the servers. Details of the implemented prototypes are presented in Deliverable D5.2 “User Profiling and Personalization”.

8. SPACE REQUIREMENTS

Most of the MApp components produce a number of events, which are eventually sent to the backend server for storage, processing, and, in several cases, use by the recommender system. It is difficult to derive an analytic formula that gives an exact estimate of the number of events that may be generated by a user over a time period. Some users may make light use of the MApp applications whereas others may make quite heavy use. Obviously, users who move a lot, and use a multitude of MApp applications simultaneously, for example the social radar to communicate with volunteers, the navigation service to get help how reach a destination or move around a city, the text lens for reading signs and translating the messages, the learning service for taking lessons, as well as a number of services that use the device's sensors to identify the user's location and the user's movements, may result into large amounts of generated events which may pose significant storage requirements. It is expected though that average users will make a more mild use of the MApp applications, resulting into moderate and, at any rate, acceptable storage requirements, according the analysis presented below.

Table 3 summarizes the data and corresponding sizes Mapp components send back to the server.

Table 3: Data Usage and Storage for each Mapp Component.

Mapp Component	**Event**	**Response size (in bytes)**	**Storage size (in bytes)**	**Interval**	**Comments**
Language learning	Usage	48	253	Every time the user exits the Mapp component	
TextLens	Usage	48	253	Every time the user exits the Mapp component	
	Upon capturing/translating a sign		253	Every time the user takes a photo of a sign and TextLens detects/translatesthe recognized text	
User Profile	Usage	48	253	Every time the user exits the Mapp component	
	GPS tracking	48	381	Every one minute and only if the user has moved 500m from last such event	Only if user has set GPS Tracking to ON from User Profile Settings
Recommendations	Usage	48	253	Every time the user exits the Mapp component	
Info	Usage	48	253	Every time the user exits the Mapp	

				component	
	Article viewed by the user	48	253	Every time the user reads a Info Article	
Augmented reality	Usage	48	253	Every time the user exits the Mapp component	
Georadar	Usage	48	253	Every time the user exits the Mapp component	
	User rates an assistance	48	509	Every time the user rates an assistance	
Navigation	Usage	48	253	Every time the user exits the Mapp component	
	Route Starts / Ends	48	637	Every time the user starts and every time navigation ends	
Places of Interest	Usage	48	253	Every time the user exits the Mapp component	
	User searches for a POI	48	253	Every time the user searches for a POI	
Serious Game	Usage	48	253	Every time the user exits the Mapp component	

8.1 AVERAGE MAPP USAGE

This section presents some rough calculations for the space required to store the events that are generated by a user community, assuming an average use of the MApp applications. We assume that according the usage pattern a user makes a 10-hour use of the MASELTOV platform per day. Therefore, as the location service generates positional events every five minutes (this duration is a configurable parameter of the platform), 12 such events will be generated per hour and 120 per day. We also assume an average use of another 5 MApp applications per day, each generating a single event per hour, for a total of 170 events per day. We can be a bit more conservative and assume a total of 200 events per day per user.

Therefore, if 200 events are produced per day per user per day, and each event requires approximately 300 bytes then the storage needs will be 60 KBytes per day per user, and 21.9 MBytes per year per user. For a community of 1000 users the space requirements are 21.9 GBytes per year, which is a rather acceptable amount of data by today's storage capacities.

Actual usage statistics will be extracted from the scheduled two-month long trial tests starting September 2014. Moreover, after receiving feedback from these trials we will implement rules for removing any unused event data (for example every two or three months) so that the total size within a year will be further reduced. For example a GPS location event that produced no recommendations would make no sense to be kept in the database.

APPENDIX A: DATABASE SCHEMA

Following is the schema of the back end database as defined in the prototype implementation, which is based on MySQL. The following schema has been produced automatically by the MySQL administration tools.

```
-- phpMyAdmin SQL Dump
-- version 4.0.4
-- http://www.phpmyadmin.net
--
-- Host: 127.0.0.1
-- Generation Time: Feb 12, 2014 at 09:17 AM
-- Server version: 5.5.32
-- PHP Version: 5.4.16

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
SET time_zone = "+00:00";


/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;

--
-- Database: `maseltov`
--
CREATE DATABASE IF NOT EXISTS `maseltov` DEFAULT CHARACTER SET utf8 COLLATE
utf8_general_ci;
USE `maseltov`;

-- --------------------------------------------------------

--
-- Table structure for table `access`
--

CREATE TABLE IF NOT EXISTS `access` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_email` varchar(80) NOT NULL,
  `user_password` varchar(100) NOT NULL,
  `user_active` char(1) NOT NULL,
  `user_fullname` varchar(50) NOT NULL,
  `user_lastlogindate` datetime NOT NULL,
  `user_lastloginip` varchar(30) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `user_email` (`user_email`),
  UNIQUE KEY `id` (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=2 ;

-- --------------------------------------------------------

--
-- Table structure for table `app_langs`
--

CREATE TABLE IF NOT EXISTS `app_langs` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `lang_caption` varchar(50) NOT NULL,
  `lang_code` varchar(2) NOT NULL,
  `is_active` char(1) NOT NULL DEFAULT '0',
  `lang_english_caption` varchar(80) NOT NULL,
  `lang_direction` enum('LTR','RTL') NOT NULL DEFAULT 'LTR',
  PRIMARY KEY (`id`),
  UNIQUE KEY `id` (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=9 ;

-- --------------------------------------------------------

--
-- Table structure for table `captions`
--

CREATE TABLE IF NOT EXISTS `captions` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `caption_id` int(11) NOT NULL,
  `caption` varchar(50) NOT NULL,
  `langid` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=289 ;

-- --------------------------------------------------------

--
-- Table structure for table `datafields_poicategories`
--

CREATE TABLE IF NOT EXISTS `datafields_poicategories` (
  `poi_category_id` int(11) NOT NULL,
  `personaldatafield_id` int(11) NOT NULL,
  UNIQUE KEY `a` (`poi_category_id`,`personaldatafield_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- --------------------------------------------------------

--
-- Table structure for table `datainfo`
--

CREATE TABLE IF NOT EXISTS `datainfo` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `type` varchar(20) NOT NULL,
  `json` varchar(300) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=10 ;

-- --------------------------------------------------------

--
-- Table structure for table `datatypes`
--

CREATE TABLE IF NOT EXISTS `datatypes` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `short` varchar(10) NOT NULL,
  `caption` varchar(40) NOT NULL,
  `datainfoid` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=10 ;
```

```
-- --------------------------------------------------------

--
-- Table structure for table `enum_captions`
--

CREATE TABLE IF NOT EXISTS `enum_captions` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `enc_id` int(11) NOT NULL,
  `enc_caption` varchar(80) NOT NULL,
  `enc_langid` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=653 ;

-- --------------------------------------------------------

--
-- Table structure for table `events`
--

CREATE TABLE IF NOT EXISTS `events` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `source` varchar(100) NOT NULL,
  `timestamp` datetime NOT NULL,
  `userid` int(11) NOT NULL,
  `receivedFromDeviceAt` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=216 ;

-- --------------------------------------------------------

--
-- Table structure for table `event_data`
--

CREATE TABLE IF NOT EXISTS `event_data` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `event_id` bigint(20) NOT NULL,
  `key` varchar(30) NOT NULL,
  `value` varchar(80) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=372 ;

-- --------------------------------------------------------

--
-- Table structure for table `fields_categories`
--

CREATE TABLE IF NOT EXISTS `fields_categories` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `langid` int(6) NOT NULL,
  `caption` varchar(100) NOT NULL,
  `order` smallint(6) NOT NULL,
  `isActive` bit(1) NOT NULL DEFAULT b'1',
  `fcid` smallint(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=25 ;

-- --------------------------------------------------------
```

```
--
-- Table structure for table `field_descriptions`
--

CREATE TABLE IF NOT EXISTS `field_descriptions` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `fieldid` int(11) NOT NULL,
  `text` text NOT NULL,
  `langid` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=233 ;

-- --------------------------------------------------------

--
-- Table structure for table `languagelearning_suggestions`
--

CREATE TABLE IF NOT EXISTS `languagelearning_suggestions` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `field_option_id` int(11) NOT NULL,
  `level` int(11) NOT NULL,
  `url` varchar(500) NOT NULL,
  `text` varchar(500) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=7 ;

-- --------------------------------------------------------

--
-- Table structure for table `personaldatafields`
--

CREATE TABLE IF NOT EXISTS `personaldatafields` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `caption` int(11) NOT NULL,
  `datatype` int(11) DEFAULT NULL,
  `datainfo` text NOT NULL,
  `mandatory` char(1) NOT NULL DEFAULT '0',
  `userEditable` char(1) NOT NULL DEFAULT '1',
  `field_category_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `caption` (`caption`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=30 ;

-- --------------------------------------------------------

--
-- Table structure for table `poi_categories`
--

CREATE TABLE IF NOT EXISTS `poi_categories` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `tag` varchar(100) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=11 ;

-- --------------------------------------------------------

--
```

```
-- Table structure for table `recommendation`
--

CREATE TABLE IF NOT EXISTS `recommendation` (
  `rec_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `rec_rectimestamp` varchar(14) NOT NULL,
  `rec_timestamp` varchar(14) NOT NULL,
  `rec_text` varchar(500) NOT NULL,
  `rec_tag` varchar(60) NOT NULL,
  `rec_userid` int(11) NOT NULL,
  `rec_expires` varchar(14) DEFAULT NULL,
  `rec_status`  enum('WAIT','SENDING','SENT','FAILED','EXPIRED')  NOT  NULL
DEFAULT 'WAIT',
  `starred` bit(1) NOT NULL DEFAULT b'0',
  `read` bit(1) NOT NULL DEFAULT b'0',
  `deleted` bit(1) NOT NULL DEFAULT b'0',
  PRIMARY KEY (`rec_id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=4614 ;

-- --------------------------------------------------------

--
-- Table structure for table `users`
--

CREATE TABLE IF NOT EXISTS `users` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_email` varchar(80) NOT NULL,
  `user_password` varchar(200) NOT NULL,
  `user_preferences` text NOT NULL,
  `last_login` datetime NOT NULL,
  `registered` datetime NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=33 ;

--
-- Triggers `users`
--
DROP TRIGGER IF EXISTS `users_before_delete`;
DELIMITER //
CREATE TRIGGER `users_before_delete` AFTER DELETE ON `users`
 FOR EACH ROW BEGIN
  DELETE FROM user_datafields_values where user_datafields_values.user_id =
OLD.id;

 END
//
DELIMITER ;

-- --------------------------------------------------------

--
-- Table structure for table `user_datafields_values`
--

CREATE TABLE IF NOT EXISTS `user_datafields_values` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_id` int(11) NOT NULL,
  `field_id` int(11) NOT NULL,
  `value_id` int(11) DEFAULT NULL,
  `value` varchar(200) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=2021 ;

-- --------------------------------------------------------

--
-- Table structure for table `wikisearchcombinations`
--

CREATE TABLE IF NOT EXISTS `wikisearchcombinations` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `treeId` int(11) NOT NULL,
  `resourceId` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=10 ;

-- --------------------------------------------------------

--
-- Table structure for table `wikisearchresources`
--

CREATE TABLE IF NOT EXISTS `wikisearchresources` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `url` varchar(500) NOT NULL,
  `desc` varchar(500) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=7 ;

-- --------------------------------------------------------

--
-- Table structure for table `wikisearchtree`
--

CREATE TABLE IF NOT EXISTS `wikisearchtree` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `text` varchar(100) NOT NULL,
  `parent` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8 AUTO_INCREMENT=11 ;

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```