Dataset Aggregator

# Prerequisites

## Background information

There are a series of documents that describe the scope, application and workflow of the application:

* **NIPN Integrated Data Repository**: This document can be found in the *annexes* folder at the top level of the documentation folder. The project started with the idea of creating a generic dataset repository that could be applied to any type of dataset, this document describes the initial idea.
* **SMART Survey Aggregator**: This document can also be found in the annexes folder. This document is a further version of the project in which the application focuses in aggregating [SMART surveys](https://smartmethodology.org/).

Both documents provide an idea of the use and function of the application. Although the second document focuses on a specific type of surveys, the principles, structures and workflow can be applied to any kind of dataset, making both documents a good base for understanding what the final application should do and how it should work.

## Database framework

[ArangoDB](https://www.arangodb.com/) is the database system upon which the application was designed. It is a multi-model database that implements a micro-services framework, [Foxx](https://docs.arangodb.com/3.3/Manual/Foxx/AtAGlance.html), which is used to implement the back-end services. You should become familiar with this database system, since the application relies upon it.

## Structure

The application is implemented with two main components: the *back-end* and the *front-end* as a web-based application.

The back end takes care of storing and serving all the data, it is implemented using the ArangoDB Foxx micro-services framework which is executed in the database itself. The goal of these services is to provide a high-level interface to data for the front-end.

The front-end has not been developed yet, the plan was to create it using [React](https://reactjs.org/), since this framework is well adapted for creating independent components that can be assembled into a working application. The other reason is that both the Foxx framework and React use JavaScript as their main language, which makes for a consistent development environment. The front-end could be developed also using other frameworks and languages, such as PHP, for instance, it should be the decision of the front-end developers to decide what makes the best choice.

# Short summary

This is a very short summary of what the application should do and how it is structured, you can consider it as the summary of the two background documents cited in the beginning and serves the purpose of laying down the principles that apply to the rest of this document.

The application should implement a repository or searchable archive of studies, their documents and the data that they represent.



The goal of this system is to provide access to the studies, their annex documents and to the raw data by allowing users to query the studies metadata and the raw data variables, retrieving two sets of results: the studies and a selection of raw data that satisfies the query conditions.

The study represents a project, survey, or other types of data collection activities that can be considered as an independent entity. The study is tagged by metadata that records when it occurred, which regions it studied, which institutions were involved and all the other information that is necessary to document the conditions, methods and scope of the project.

All annex documents - in the case of surveys these could be the questionnaires, reports and original datasets – should be uploaded and linked to the study, so that it can be considered as an archive of all its information.

Studies will have one or more datasets that represent their data. The dataset exists in two forms: as the original file – Excel, Stata, SPSS, etc. – which will be uploaded as an annex file of the study, and as the data it contains that we will call raw data here. Raw data will be stored in a collection that corresponds to the data domain of the dataset. If we use SMART surveys as the example, all the SMART survey raw data will be stored in a single data collection which represents the domain of SMART surveys. The goal of this structuring is to allow searching all SMART survey data and retrieving aggregated data selections that can be downloaded by users to perform further data analysis or summaries. Data collections can be organised in ontologies to allow more than one data domain, for instance you could have a census collection that holds the data from all census datasets, this structuring allows the system to handle different types of data[[1]](#footnote-1).

The central component of the system is the data dictionary, its role is to provide the definition and documentation for all the fields of all records stored in the database. When importing raw data from a dataset, the first function to perform is to harmonise the variables of the raw data with the data dictionary, so that each column of data in the raw data table corresponds to a descriptor of the data dictionary[[2]](#footnote-2).

Once studies have been registered, their annex documents uploaded, and their data harmonised with the data dictionary, it becomes possible to perform a set of queries whose results will be a selection of studies and a selection of raw data. These two sets of results could be presented in two separate panes. The studies could be chosen and downloaded as a zipped archive in which the study metadata record could be provided as an Excel document along with all the other annex documents. The raw data could also be downloaded as a CSV file or a file of some other format. As the application matures, statistical functions could be applied to the raw data selections in the back-end to provide further functionality and features to the application users.

# Database structure

The database contains a series of predefined collections (tables in the traditional relational nomenclature), each serving a specific purpose:

* descriptors: This collection contains the variables definitions, each record represents a variable, its type, label, description and all the other information that is necessary to document and validate data associated with that variable.
* terms: A term is an item that has a code and a series of descriptive properties in several languages whose code is used as a reference. Terms are organised as ontologies (tree or graph structures) and implement all the relational structures of the data dictionary, such as controlled vocabularies (or enumerations), forms, ontologies, etc.
* schemas: This collection contains all the links between the data dictionary elements, parent-child relationships in controlled vocabularies and forms, as well as relationships between terms and their instances are recorded in this collection.
* users: This collection contains all the user records.
* hierarchy: This collection contains all the edge documents that relate users with other components of the data dictionary, for instance, such as the relationship between a user and its manager.
* studies: This collection contains the studies metadata records.
* annexes: This collection contains the study annex document records[[3]](#footnote-3).
* toponyms: This collection contains a set of toponyms which are related to terms defining geographic units. Toponyms are not needed by the application but represent a multilingual repository of geographic locations and administrative units, along with their nomenclature.
* shapes: This collection contains a set of GeoJSON shapes related to toponyms.
* edges[[4]](#footnote-4): This collection contains the relationships between toponyms, shapes and terms.
* errors: This collection contains error type definitions. The elements of this collection can be compared to the JavaScript Error object name.
* messages: This collection contains errors and other messages, the errors can be compared to the JavaScript Error message property, while the other entries are used to store the descriptions of the services in several languages.
* settings: This collection contains the settings for the application, it currently contains one entry that indicates the status of the application.
* sessions: This collection contains session data, it can be used by the front-end to store session user specific states.
* logs: This collection contains the log of the services, it records what services were called by which user and the user status at the call time.
* groups: This collection was supposed to contain user groups. User groups are not implemented, although they are earmarked in the code.
* smart: This collection should contain the SMART survey raw data as SMART datasets are harmonised and added to the database.

# Data Dictionary

The main component of the application is the data dictionary, all fields of all collections *must* have an entry in the data dictionary, this means that all data elements are defined, documented and available.

## Common default fields

The key fields of all collections follow the ArangoDB standards:

* **\_id**: This field contains the unique identifier of the record within the *database*. It is the concatenation of the collection name, followed by a ‘/’ token and the value of the \_key field. This field will generally never be set programmatically since the database will automatically set it, but it is required when referring to a record belonging to an unknown collection. Once set, the value cannot be changed.
* **\_key**: This field represents the unique identifier of the record within its *collection*. This will be the value used to identify a specific record in a collection. In some cases, this field will have to be set explicitly, in other cases it is computed from other fields in the record. Once set, the value cannot be changed.
* **\_rev:** This field represents the record revision. It is never explicitly set, but it is used when updating or replacing a record: if the record selector contains a revision and that revision is different than the current record revision, the operation will fail. This is useful to ensure nobody modifies a document that is currently under modification.

All collections, document and edge, feature these fields and they have the same function in both types of collection.

Edge collections have two additional fields:

* **\_from**: This field represents the relationship source node \_id. Once set, the value cannot be changed.
* **\_to:** This field represents the relationship destination node \_id. Once set, the value cannot be changed.

## Global and local identifiers

Some collections, in particular terms and descriptors, use an additional identification scheme, this scheme uses the following fields:

nid: This field represents the namespace of the identifier. It is the \_id of another record. The referenced record must either implement this identification scheme or have the global identifier field (gid). This field is generally not required, since there would not be any way to define a namespace.

lid: This field represents the local identifier. The value must be unique within its namespace. This field is required and cannot be empty[[5]](#footnote-5).

gid: This field represents the global identifier. The global identifier should be unique among all namespaces. It is a computed field that is set by concatenating the gid of the record referenced by the nid (namespace), followed by a colon (:) and ending with the current record’s nid (local identifier). If the record does not have the namespace, the global identifier will be its local identifier.

For instance, suppose we want to create a namespace in the terms collection:

{

\_id: “terms/ISO”,

\_key: “ISO”,

lid: “ISO”,

gid: “ISO”

}

This namespace could be used to create another terms record such as:

{

\_id: “terms/ISO:ITA”,

\_key: “ISO:ITA”,

nid: “terms/ISO”,

lid: “ITA”,

gid: ISO:ITA”

}

This scheme is used in particular for controlled vocabularies where the code may be ambiguous depending on its domain. For instance, we have the it code for *Italy*, the country and we also have the it code for *Italian*, the language: to be able to uniquely identify both codes we make use of the namespace.

This means that all controlled vocabulary elements in the database will have the full code, namespace and local identifier; this also means that given a local identifier we can easily select all entries that might be a match, something useful when importing data from different sources.

## Data types



Data types are defined as a controlled vocabulary of term elements. This controlled vocabulary defines the format, scope and domain of values belonging to a specific descriptor. The data type identifier is the \_key of the term, which is copied from the term global identifier, see above.

By standard, no values can be null, a variable may either have a non-empty value, or not be there.

Data types are organised as a hierarchy and implemented as branched edges where the branch is the specific data type and the predicate is terms/:predicate:type-of. The first level elements, those in orange in the above diagram, represent the data domains. Each domain has a specific set of modifiers, that are essentially constraints applied to the domain. For instance, the text domain includes a modifier which constrains the minimum and maximum number of characters that the text may have; numbers have a similar modifier that restricts the value to a range.

Some of these modifiers are implemented in the data type, other are implemented in the descriptor. For instance, the :type:value:term data type represents a reference to the \_key field of a term, the data type sets terms as the default collection; descriptors of that type may include a list of controlled vocabularies to which values of that type must belong.

To create a specialised data type, you derive from a type and modify the inherited modifiers. For instance, the string data type sets a constraint limiting the maximum number of characters to 254. Or you implement a descriptor and override the modifiers in the descriptor.

Besides reading the description below, I advise you to look at the definition of the data type terms in the terms table, the description fields contain further information to get a better idea of their function.

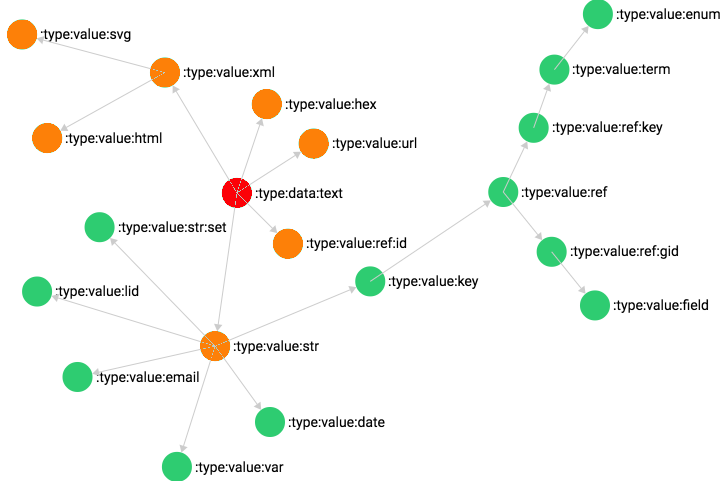
### Base data types



* **[:type:data:any**]: This data type represents a value that can take *any* type. Values of this type will not be validated, since they can take any value. This data type has no derived types.
* **[:type:data:bool**]: This data type represents a true/false *Boolean* value, it can be represented in a form as a radio button or checkbox. This data type has no derived types.
* **[:type:data:text**]: This data type represents *text* data, it is by definition utf8 characters and can take any size. Because of the lack of size limits this variable can be represented with a scrollable text area in a form. This base type can have the following modifiers: minimum and maximum number of characters, regular expression and range.
* **[:type:data:numeric**]: This data type represents a number that can either be an integer or a float[[6]](#footnote-6). This base type can have the following modifiers: range and the number of decimal positions[[7]](#footnote-7).
* **[:type:data:list**]: This data type represents a list or array of items of undefined type. This base type has one modifier which determines the minimum and maximum number of elements.
* **[:type:data:struct**]: This data type represents an object or structure whose elements must follow the data dictionary rules. This means that when the validation script encounters such a value, it will recursively validate its elements. This base type does not have modifiers.
* **[:type:data:object**]: This data type represents an object or structure whose elements are parsed as a key and value pair. This base type has two modifiers, type-key:type and type-value:type, that can be used to indicate respectively the data type of the pair key[[8]](#footnote-8) and value.

### Text data types

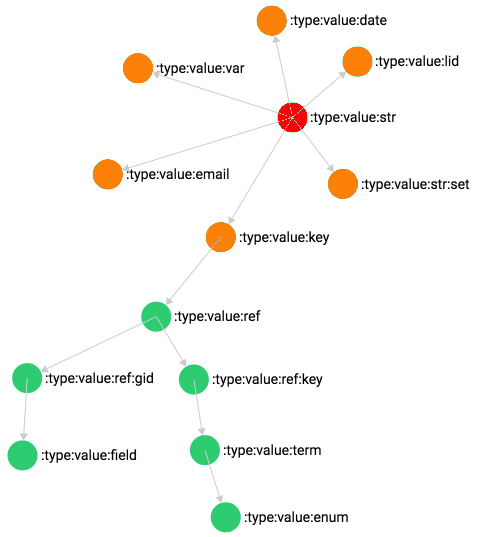
Text data types represent generic character data, ArangoDB treats all text as a sequence of UTF8 characters.



* [**:type:value:hex**]: This data type represents a hexadecimal value.
* [**:type:value:url**]: This data type represents an URL or internet link.
* [**:type:value:xml**]: This data type represents an XML text.
* [**:type:value:svg**]: This data type derives from the XML type and represents an SVG image.
* [**:type:value:html**]: This data type derives from the XML type and represents an HTML text.
* [**:type:value:str**]: This data type derives from the text type and represents a string that is limited to 254 characters. This type is used for text variables that can be indexed and used as object identifiers.
* [**:type:value:ref:id**]: This data type derives from the text type and represents the \_id field value of another record. This data type is used to directly refer to another object.

### String data types

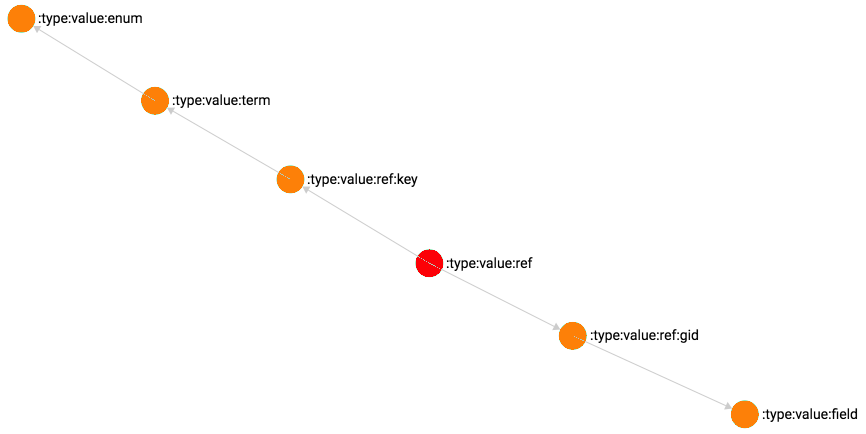
This set of data types derive from the string type, these will be generally used to represent short strings of data such as titles, labels and names.



* **[:type:value:key**]: This data type is used for strings that are used as record keys, the string cannot contain spaces, it allows a subset of punctuation and it must be non-empty and up to a length of 254 characters.
* **[:type:value:var**]: This data type is used for strings representing variable names, the string must be alphanumeric, non-empty and up to 64 characters long; it allows the use of underscores.
* **[:type:value:lid**]: This data type is used for strings representing local identifiers (lid), the string cannot contain spaces, it allows a subset of punctuation and it must be non-empty and up to a length of 64 characters.
* **[:type:value:email**]: This data type is used to represent e-mail addresses.
* **[:type:value:date**]: This data type is used to represent a string date with optional month and day in the YYYYMMDD format.
* **[:type:value:str:set**]: This data type is used to represent a list of unique strings.

### Reference data types

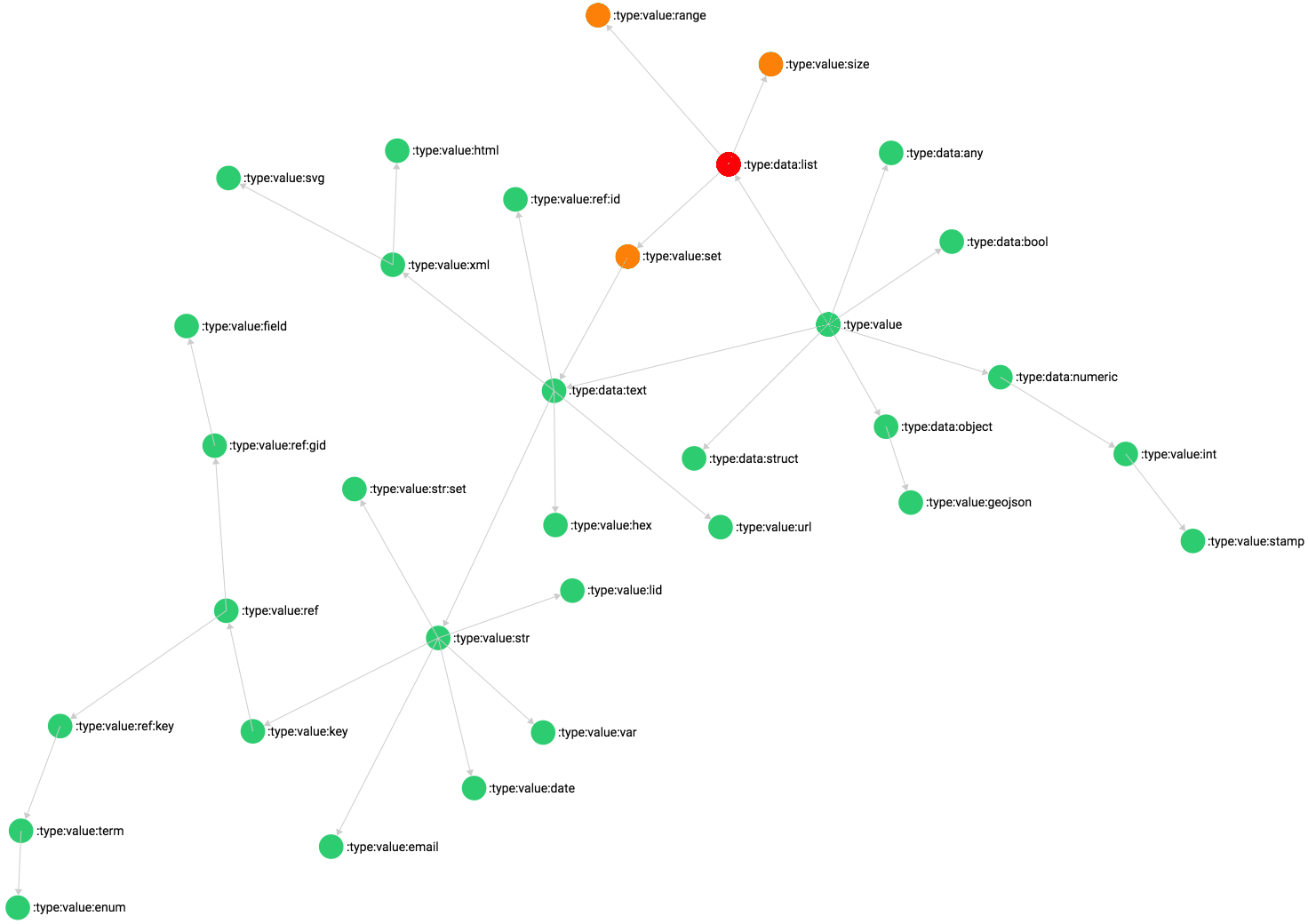
Reference data types are used for fields that reference another object through its \_key or gid field[[9]](#footnote-9).



* **[:type:value:ref:gid**]: This data type represents a reference to the gid field of another object.
* **[:type:value:field**]: This data type derives from the :type:value:ref:gid type and sets descriptors as its default collection. This data type is used to reference data fields or descriptors.
* **[:type:value:ref:key**]: This data type represents a reference to the \_key field of another object.
* **[:type:value:term**]: This data type derives from the :type:value:ref:key type and sets terms as its default collection. This data type is used to reference terms.
* **[:type:value:enum**]: This data type derives from the :type:value:term type, as its parent, it expects the value to be a reference to a term, except that the referenced term must define a controlled vocabulary[[10]](#footnote-10). This data type is used to reference enumerations that apply to a controlled vocabulary field.

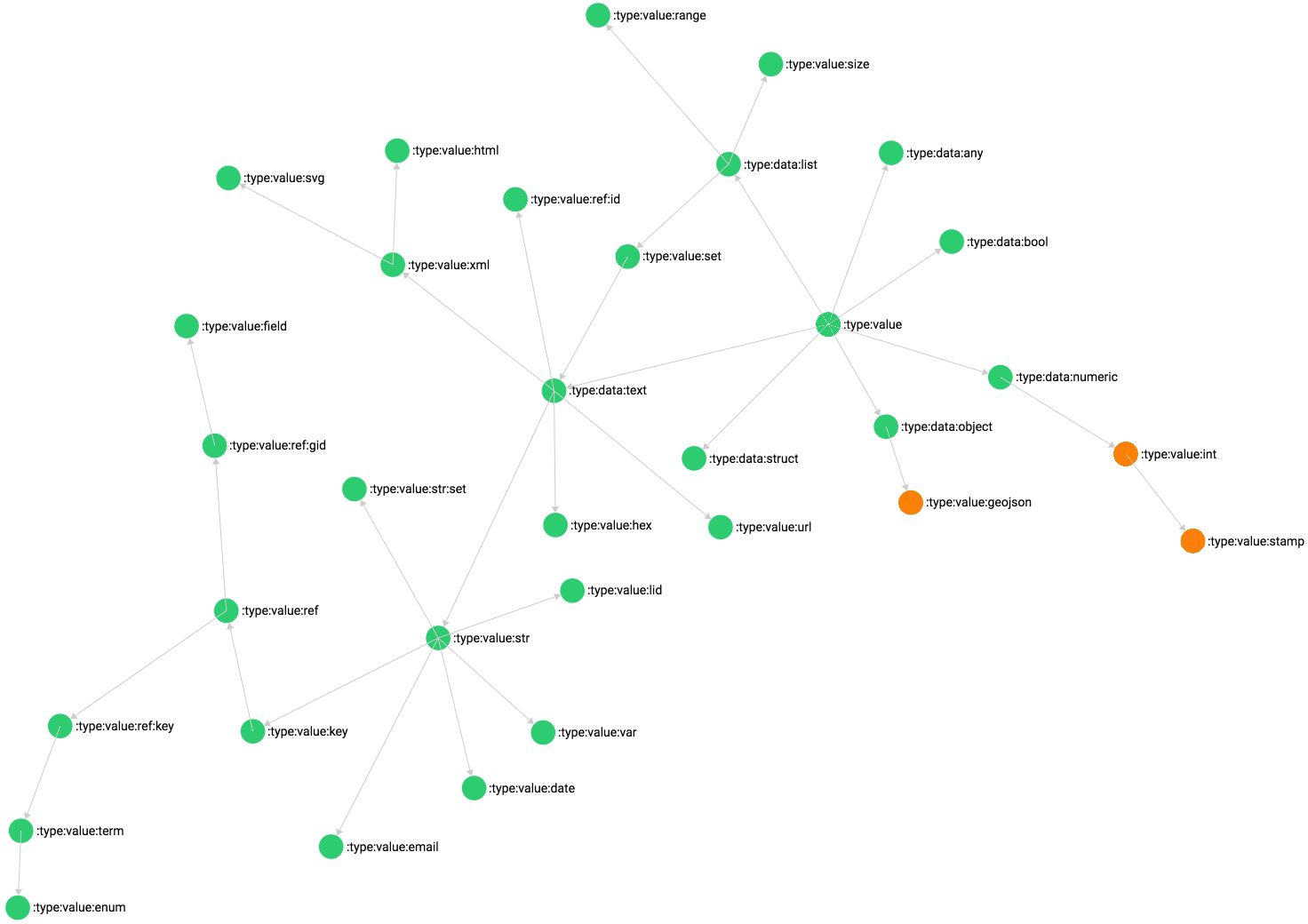
### List data types

List data types represent list of elements of a specific type. Although descriptors have a property that can be used to indicate that the variable contains a list of elements of the descriptor data type, list data types are inherently lists.



* **[:type:value:set**]: This data type represents a list of unique elements.
* **[:type:value:range**]: This data type represents a range, it is implemented as an array of 4 elements which are the minimum, maximum, include minimum and include maximum. The first two elements are numbers, the last two elements are Booleans which indicate whether or not the limits are included in the range. The first Boolean refers to the minimum bound, the second to the maximum bound.
* **[:type:value:size**]: This data type represents a size range, it is structured as the previous data type, except that it indicates a size, rather than a value range. It is used to restring string length and list elements count.

### Other data types



* **[:type:value:int**]: This data type represents integer values.
* **[:type:value:stamp**]: This data type derives from the integer data type and is used to represent time stamps.
* **[:type:value:geojson**]: This data type represents a GeoJSON structure, it can be used to store and validate geometric shapes.

## Terms

Terms are used to implement elements of controlled vocabularies, forms, types, predicates and all other items that represent root elements of ontology structures. All terms reside in the terms collection.

Terms implement the local/global identifier scheme and the value of the global identifier (gid) is copied by default to the object’s \_key. Once a term is inserted, the nid, lid and gid field values become locked.

* **\_key**: This field contains the same value of the gid field, it represents the global identifier of the term.
* **nid**: This field contains the \_id reference to the term that represents the term’s *namespace*. All terms should have a namespace, except terms that define namespaces.
* **lid:** This field contains the term local identifier unique within its namespace (nid). This field is required.
* **gid:** This field contains the term global identifier, it is a required computed field described in the “Global and local identifiers” section at the beginning of this document.

Terms, in their minimalist form, will contain the identifier fields and a set of text description fields that represent the name, description and other information. These fields are:

* **label**: This is the name, title or label of the term, whenever a term is referenced this field can be used as a label for the term. *This field is required*.
* **definition**: This field represents the definition of the term, while the label should not be long, this can be a long text and should provide the necessary information elements. This field could be used for an info button.
* **description**: This field can be used to expand the definition with a more extensive description.
* **note**: This field can be used to add notes and comments.
* **example**: This field can be used to provide examples, if that is relevant.

These description fields are multilingual, and they are implemented by using the :type:data:object base data type and setting the specific key and value modifier data types in the descriptor. The property names of the object are language codes and the values are text, for label and definition, or html for the other fields. This is the default structure for all multilingual content.

Terms also contain a set of properties that represent alternative identification or keywords:

* **var**: This field represents a variable name, it can be used as such when referring to a term. In practice, this field is set in all default terms, there is a service in the application that generates a JavaScript file in which the \_key and var fields can be consulted as a dictionary: in the source code I always refer to the var value when referencing terms, this allows me to change the global identifier without needing to revise the source code.
* **sym**: This field represents the term symbol, it is a string that represents the acronym of the term, such as cm. for centimetres.
* **syn**: This field is an array of synonyms for the term, by default it contains the local identifier and the variable name, additional used synonyms can be added to aid in selecting the right term.
* **keyword**: This is a set of keywords that can be applied to the term.
* **type-cast**: This field is a controlled vocabulary that is used only in data types for validation purposes. The value indicates what type casting is to be applied to values of the specific data type before performing the validation. For instance :rule:castNumber indicates that the value must be cast to a number.
* **type-custom**: This field is a controlled vocabulary that is used only in data types for validation purposes. The value indicates what custom validation procedure is to be applied to values of the specific data type in addition to the default procedures[[11]](#footnote-11). For instance :rule:customHex indicates that the value should correspond to a hexadecimal number.
* **collection**: This field is a controlled vocabulary that references terms defining collections. It is used in instance controlled vocabularies to indicate in which collection the instance should be stored.
* **instance**: This field is a controlled vocabulary that references the instances controlled vocabulary. I actually forgot what its use is and there is only one term that has this field, so let’s ignore it for the time being...
* **traversal**: This field is a controlled vocabulary that represents graph traversal directions, it is used exclusively in predicate terms to indicate the direction of the predicate. For instance, the :predicate:category-of predicate has the :type:traversal:is value: this means that the relationship source *is* a category of the relationship destination. This value is useful to determine in what direction a graph should be traversed according to the predicate.
* **length**: This field is a value of :type:value:size type that is used exclusively in data type terms as the modifier indicating a character count range restriction.
* **regex**: This field is a string used exclusively in data type terms to provide the regular expression for validating text data.
* **deploy**: This field is required both in terms and descriptors, it is a controlled vocabulary that indicates what deployment status of the current object:
  + **[:state:application:embedded**]: It is an object implemented by the database; only descriptors can fall under this category.
  + **[:state:application:default**]: It is default object.
  + **[:state:application:standard**]: It is an object that implements a standard.
  + **[:state:application:user**]: It is a user-defined object.
* **level**: This field is a number that is used as a priority or hierarchical indicator, lower numbers represent higher priorities or top hierarchical levels.
* **class**: This field is a controlled vocabulary that indicates the class of the term, it is used in the instances controlled vocabulary to indicate the class of the instance.
* **store**: This field is a controlled vocabulary that indicates the collection type, it is used in the collection controlled vocabulary to indicate whether the collection is document or edge.
* **instances**: This field is a controlled vocabulary set of values that indicates which instances the current term has taken. For instance, a term may represent a controlled vocabulary, in that case this field would contain :instance:enumeration; the same term may, however, also be used as a controlled vocabulary selection element, in that case the field would also contain :instance:selection. This field is used to select terms according to how they were used.
* **info**: This field is an array of URLs providing information resources.
* **schema**: This field is an URL pointing to a resource that represents the term schema. For instance, the JSON schema of ISO standards is published in the web, the term that defines the corresponding ISO controlled vocabulary could also include this information.

## Descriptors

The descriptors collection contains all variable definitions, the \_key field value represents the record field name and the other fields contain the data type, definitions and all other information necessary to document and validate values belonging to that variable.

Although the descriptors collection implements the global/local identifier scheme, in this case the global identifier is not copied into the \_key field, this is because the value of this field becomes the name of the record field which may pose problems both because of the characters used and because the value may become long, making the structure of the record inefficient[[12]](#footnote-12).

Since descriptors share most of their properties with terms, we shall only include here those fields that terms do not have, or that have a different function; note that the fields described below are descriptors, which means that the data dictionary uses itself for its definitions.

* **\_key**: This field contains the unique identifier of the descriptor. The value will become the field name associated with that variable definition.
* **nid**: This field contains the \_id reference of the term or descriptor that represents the variable’s *namespace*, this field is required for descriptors.
* **lid:** This field contains the local identifier unique within its namespace (nid). This field is required.
* **rank:** This field is a controlled vocabulary associated with users, it represents the user rank. In descriptors it is used to restrict usage to users having a rank with a level lower or equal to the current session user’s.
* **kind:** This field is a controlled vocabulary that defines the data kind, it indicates whether the value is qualitative, categorical, ordinal, discrete or quantitative.
* **type:** This field is a controlled vocabulary that defines the data type, these have been described earlier.
* **format:** This field is a controlled vocabulary that defines the data format, it can be used to indicate that the descriptor contains a scalar, a list or a set of values of the descriptor’s data type.
* **terms:** This field is a controlled vocabulary that applies exclusively to descriptors of type :type:value:term or :type:value:enum, it is a controlled vocabulary set of terms that define controlled vocabularies. In :type:value:term descriptors it limits the scope of the term reference to one of the controlled vocabularies belonging to the list. The field is used to restrict the choice of term references to a defined set of values.
* **fields:** This field is identical to the terms field above, except that the elements or choices of the controlled vocabulary will not be terms, but gid descriptor references. This field is used to provide a choice of fields.
* **type-key:** This field is an object of type :type:data:struct that is used exclusively by :type:data:object data type descriptors. It serves the purpose of defining the scope and type of the object keys. It *must* contain the type field and any other field that is used to define the data type. For instance, multi-language strings have type = :type:value:term and terms holding the list of controlled vocabularies that categorise languages. The data type applies to all property names of the object.
* **type-value:** This field is an object of type :type:data:struct that is used exclusively by :type:data:object data type descriptors. It serves the purpose of defining the scope and type of the object property values. It *must* contain the type field and any other field that is used to define the data type. For instance, multi-language strings have type = :type:value:str or another text type variant. The data type applies to all values of the object.
* **size**: This field is a value of :type:value:size type that is used exclusively for lists and sets as the modifier indicating an element count range restriction.
* **size**: This field is a value of :type:value:range type that is used as the modifier indicating the valid range for values of that descriptor.
* **regex**: This field is a string that represents the validation regular expression, it is used by descriptors of type text.
* **unit:** This field is a controlled vocabulary that defines the descriptor values unit.
* **units:** This field is a controlled vocabulary set of unit elements that lists all the other units in which values could be encountered. This field can be useful when harmonising datasets: it can be used to provide users with a choice of unit conversions.

### Users

The users collection contains the user records. Users have the following fields:

* **\_key:** The user key is assigned by the database. All references to users will use this field, but publicly users are referred to by their user code.
* **name**: The user full name.
* **email:** The user e-mail address.
* **language:** The user’s preferred language. This field is a controlled vocabulary of language codes, :enum:language, which indicates the language in which the user wants to interact in. Note that the above mentioned controlled vocabulary contains the *installed* languages, not the available languages.
* **username**: The user code, it is the login code; if omitted, it defaults to the user email.
* **rank**: This is a controlled vocabulary that indicates the rank of the user. It is not yet used anywhere, but might be used as the hierarchical category of the user where higher levels provide access to lower levels:
  + :rank:system: This rank would be the highest, providing access to all resources.
  + :rank:default: This rank should give access to *default* terms, descriptors and others.
  + :rank:standard: This rank should give access to *standard* terms, descriptors and others.
  + :rank:standard: This rank should give access to *standard* terms, descriptors and others.
  + :rank:user: This rank should give access to *user-defined* terms, descriptors and others.
  + :rank:guest: This rank represents the *guest* rank, that is, the lowest level.
* role: This is a controlled vocabulary set, it contains the codes that indicate the allowed operations for the user:
  + :role:user: Can create and manage other users.
  + :role:batch: Can initiate and execute batch jobs.
  + :role:upload: Can register studies and upload annex files and datasets.
  + :role:meta: Can manage metadata, the scope is limited by the user rank.
  + :role:clean: Can harmonise datasets.
  + :role:suggest: Can create data dictionary elements with a temporary status. Suggested items should be reviewed by the data dictionary curator who will then make these elements adopted.
  + :role:dict: Can curate the data dictionary.
  + :role:commit: Can decide when a dataset has been harmonised and commit it to its final collection.
  + :role:query: Can query the data contents of the database.
  + :role:download: Can download the data contents queries.
* auth: This is an object that contains the user’s authentication data, it should be treated as a black box and never be made accessible to users.

## User management

There are two types of users: the system administrator and regular users. The system administrator can only be created is there are no other users, it must be the first user in the system. The creation process is the same as for other items:

1. The front-end requests a form, optionally providing initial data.
2. The front end receives the form record optionally including the provided data.
3. The front end presents the form to the user and returns the data to the back end service which records the data in the database.

The system administration creation workflow follows these steps:

1. Front-end requests system administrator sign-up form by providing the system administration authentication token. (/user/signup/admin/form).
2. Back-end sends the form.
3. Front-end parses the form and creates the user interface form.
4. Front-end presents the form to the current user (not yet an actual user, since there are still none).
5. The user fills the necessary information.
6. Front-end sends back an object with two properties: token and data: token contains the same token sent in the first call and data should contain the form data. (/user/signin/admin).
7. The back-end validates the form data and creates the system administrator, logging in the user at the end.
8. The back-end returns the user record as the response.

Once the system administrator has been created, it is possible to create regular users. Regular users must be created by users that have the :role:user role. The user that creates another user becomes its manager. A manager has access to all the users it creates and to all users created by users it created. It is a tree structure in which the root, the system administrator, has access to all the users. The steps are the following:

1. Front-end requests user sign-up form by providing the user authentication token. (/user/signup/user/form).
2. Front-end posts the form, gets the form data and sends the data back to the (/user/signup/user) service.
3. The back-end creates a temporary user record and sends back a string token to the front-end.
4. The front-end creates an invitation e-mail that contains a link including the token and sends it to the user (the user e-mail field is required).
5. The user presses the link which is received by the front-end.
6. The front-end creates a request to the (user/signin/user/form) service with two parameters: token and encoded. The first parameter is the same token sent to the (/user/signup/user) service, while the second parameter is the response received by this service.
7. The back-end uses the token to authenticate the service and encoded to retrieve the temporary user record previously created by the (/user/signup/user) service. The service then creates a form record embedding the data of the temporary user and sends it back to the front-end.
8. The front-end presents the form to the user who updates its personal information, including the password. The front-end sends the form data to the (user/signin/user) service along with the same token parameter.
9. The back-end validates the form data, updates the user record setting it as active and logs the user in. It then returns the user record to the front-end.

## References and Relations

Fields values may be references to another record of another collection. It is the equivalent of a relation in a relational structure. The value of the field is a reference to the record that describes that field. In this application we use this especially for controlled vocabularies: for instance, you may have a “country” field that contains the key of the record that defines that country. This kind of relationship, that we call here *reference*, is used whenever you are referencing a single object and the relation links two vertices.

References are implemented in three ways:

* \_id reference. The value of the field is the \_id of the referenced record. In this case, since the \_id contains the collection to which the referenced record belongs, it is the only bit of information needed.
* \_key reference. The value of the field is the \_key of the referenced record. Since the collection is unknown, this kind of reference is indicated by a specific data type. For instance, the :type:value:term data type indicates that the value of the field must be the \_key of a record in the terms collection. This is the data type of all controlled vocabulary variables and currently the only data type of this kind.
* gid reference. The value of the field is the gid of the referenced record. As for the previous type of reference, it is the data type that indicates the referenced collection. This data type is used to reference descriptors, the :type:value:field is currently the only data type of this kind[[13]](#footnote-13).

When we want to consider a relationship that connects more than two vertices, such as the paths that connect a set of cities, we use a different way of relating elements: we use *edge collections*. Each record in this type of collection contains the source and destination vertex references and other information that quantifies and qualifies the relationship. Since ArangoDB implements natively directed graphs, this mechanism allows us to consider a hierarchical structure as a single entity. Controlled vocabularies, forms, data types and many other concepts are represented in this way.

A relationship is uniquely identified by its *significant fields*, these will include the \_from and \_to fields, plus any other field that uniquely identifies the relationship. In this application we apply a specific identification scheme to prevent having to create a unique index for these significant fields. We concatenate the significant fields, separated by a TAB character, and hash the result, this value is set in the edge \_key field, which is by definition unique.

The application implements three types of edge:

* ***Edge***: This type of edge has as significant fields the \_from, \_to and predicate fields. The predicate represents the type of relationship or its verb, it is a controlled vocabulary. All edges in the application *must* have at least these three fields as significant. The \_key is the MD5 hash of these fields.
* ***Branched edge***. Suppose you have a hierarchical controlled vocabulary of geographic locations. You would have as root the term that defines the controlled vocabulary. As the first level children of that term you would have the continents, which would have the regions as children, which would have countries as their children, which would have sub-national administrative units as children and so on. If you want to implement a subset of that tree with the first type of edge, you would be forced to duplicate each of these edges to include only those elements in which you are interested in. This could become tedious, especially if the edge contains a lot of other data that characterises the relationship, so this type of edge can be used to re-use existing edges while keeping track of which branches they belong to. This edge adds the branches field which is an array containing the list of branches that use this relationship. This field is not significant, meaning that it is not used to compute the edge key, but it is used when traversing the graph to only follow paths belonging to a specific branch: you start with the origin vertex and follow all edges that have the predicate you are interested in and that have in their branches field the branch of interest. Controlled vocabularies use this type of edge, the branch represents which specific controlled vocabulary tree you want to traverse in the graph.
* ***Attribute edge***. This type of edge adds the attributes field to the significant fields. This field is an array of controlled vocabulary elements which is sorted just before computing the edge key, so these attributes become integral part of edge identification.

Significant fields cannot be modified once they are created. For instance, the branches field in branched edges is dynamic and elements are added or deleted in time, the attributes field in attribute edges is instead immutable.

## Enumerations

By enumerations we mean hierarchical controlled vocabularies or ontology structures. An enumeration is a tree structure in which the elements can either be valid choices or category containers.

For instance, suppose we have a controlled vocabulary of countries:



All the vertices of this graph are Term objects.

COUNTRIES represents the controlled vocabulary definition, it will contain the title, description and other information that refers to the enumeration as a whole.

CHINA, USA and COLOMBIA are the valid *choices* of the controlled vocabulary, while the other vertices are *categories* used as containers to group elements under different categories for organisation or browsing purposes (usually to organise large lists of elements).

The arrows represent Edge documents stored in the schemas collection that contain the \_from and \_to references to the vertex terms and the predicate which determines what the \_from vertex represents: a *choice* or a *category*.

In controlled vocabularies the predicate that determines a valid *choice* is terms/:predicate:enum-of, while the predicate that determines a category is terms/:predicate:category-of; in the diagram the first is red and the second is blue.

Such a structure can be implemented in the user interface like a directory, where folders are *categories* and files are *choices*.

This type of structure is used to implements all the elements of the data dictionary and should be used to implement the dictionary user interface. For instance, forms are organised in the same way, except that the *choice* predicate is terms/:predicate:field-of and the *category* predicate is terms/:predicate:form-of; this last predicate is used to include a sub-form, as the terms/:predicate:category-of predicate could be used to include a sub-controlled vocabulary.

Whenever a user wants to select something, by using such a category/choice structure, you can guide the user to the choice by directing the search through a logical path.

These structures are implemented by the three types of edges described at the beginning of the document. Branched edges are used both for controlled vocabularies and for forms.

Suppose you want to add a new controlled vocabulary to the above example by using the AMERICAS hierarchy and adding a EUROPE hierarchy.



As you see in the diagram, the Americas hierarchy is used by the CUSTOM enumeration, this is performed through the branches field in the edges. This field is present in all edges of the structure, it is an array that records all the enumerations that use the edge.

In the first example all edge branches will have one element referencing the COUNTRIES enumeration. In this example we add the CUSTOM reference to all edges under the AMERICAL hierarchy, since it is used by both enumerations; besides adding the new vertex terms, obviously.

So, to traverse a specific enumeration you follow all the graph edges that have the enumeration reference, COUNTRIES or CUSTOM, in the edge branches and the desired predicate. This allows you to start at any level of the hierarchy and traverse it in any direction; it also allows you to re-use the contents of common edges.

The edge \_key is computed by concatenating the edge significant fields separated by a TAB character. As described at the beginning of the document, there are three types of edge objects:

* Edge and Branched Edge. Their significant fields are \_from, \_to and predicate. This means that there can only be one edge with that combination of fields. For instance, the edge that connects a user to its manager has the :predicate:managed-by predicate. Branched edges work in the same way, but they have a field, branches, that is used to mark the *branches* that pass through the edges. As a rule, if more than one path may traverse a specific edge, the edge will be branched, if not, it will be a plain edge.
* Attribute Edge. This type of edge adds an attributes field to the significant fields which is an array of terms that is sorted before the edge \_key is computed, so that the order of the attributes is irrelevant.

In general, if an edge has the branches field it will be a branched edge, if it has the attributes field it will be an attribute edge and if it has neither it will be a plain edge.

Note that edge types depend on the predicates that are used, it means that controlled vocabularies and forms, for instance, will only use branched edges and you cannot mix edge types. This means that by knowing which predicates a relationship can take, you will also know what kind of edge will be used:

* :predicate:enum-of and :predicate:category-of: These predicates indicate a controlled vocabulary that uses *branched edges*.
* :predicate:field-of and :predicate:form-of: These predicates indicate a form structure that uses *branched edges*.
* :predicate:type-of: This predicates indicates a data type hierarchy that uses branched edges.
* :predicate:managed-by: This predicate relates a user to its manager and is a plain edge.
* :predicate:shape: This predicate relates toponyms to shapes, it relates GeoJSON shapes to a toponym. This is an attribute edge where the attributes field contains a set of qualifiers that categorise the shape type. For instance, a toponym may have more than one shape associated with it, the attributes field should indicate the type or function of the shape. These edges can be found in the edges collection.
* :predicate:endorse: This predicate is used to indicate the default or preferred value of a controlled vocabulary element. For instance, countries may be defined using one, two and three letter codes, each of these standards are valid and can be found in the database, however, it is preferable to use a single standard, the three letter code in our case, so that there is just one code referencing the country. This will be a plain edge type and will relate the synonym code with the default code, so that when you encounter a valid code you should check if that code points to another code with this predicate: if that is the case, you should replace the value with the pointed to one. This mechanism can be used to enforce a specific standard in an automatic way.
* :predicate:instance: This predicate is used to indicate the instance of a controlled vocabulary element. Suppose you have the term that defines the ISO code for Italy, if you have a record in the database that represents an instance of Italy, such as a toponym record containing extensive information on that country, you can have an edge stemming from the term and pointing to the toponym with this predicate. This mechanism is used to relate terms with the actual objects they refer to. This uses the plain edge type.
* :predicate:used-by: This predicate is used to indicate that one vertex is used by the other vertex. Currently it is used to relate currencies to toponyms, that is, to indicate which currencies are used where. This uses a plain edge type. This mechanism can be used when a single field with a reference is not enough to express the full relationship, using an edge allows you to add attributes and other data qualifying the relationship.
* :predicate:managed-by: This predicate is used to link a user to its manager. This uses a plain edge type. A user *must* have a manager, when you delete a user, all users that the deleted user manages will be put under the user that manages the user that will be deleted, this way everybody has a manager. It also allows adding further information to the relationship, a thing that cannot be done with a relational field.
* :predicate:registered-by: This predicate is used to link a user to the study he/she registered, this relationship makes the user the manager of the study and all its related elements. This uses a plain edge type. This mechanism is used for the same reason that manager/user relationships use edges: a user *must* have a manager and a study *must* also have a manager. When you delete a user, all studies that the deleted user manages will be put under the user that manages the user that will be deleted, this way all studies have a manager.
* STD:geo:neighbour: This predicate is used to link neighbouring geographical toponyms, it uses the plain edge type. This mechanism is used to get, for instance, the list of neighbouring countries.
* STD:geo:unit-of: This predicate indicates that one vertex is a unit of another vertex. This mechanism is currently used to

## Forms

Forms behave exactly as controlled vocabularies, while controlled vocabulary edges have only *terms* as vertices, forms will have either *terms* or *descriptors* as their vertices. The terms/:predicate:field-of predicate indicates that the \_from is a descriptor, while the terms/:predicate:field-of predicate indicates that the \_from is a term representing a sub-form.

Forms use branched edges as their relationships, the branches field contains the list of forms that use the specific edge. The way the data dictionary is structured allows using a single descriptor in different contexts, for instance, the label, definition and description descriptors are used I almost all types of objects, but their specific function may be different depending on where they are used. For instance, the label descriptor is used in descriptors and studies, except that while in descriptors a label is a label, in studies it is used as the title of the study. In order to allow *customising* default definitions according to the context in which they are used, edges have a mechanism to overload the default definitions of descriptors.

This behaviour is transparent if you are using services to retrieve a form structure, but you should be aware of how it works. Supposing we want to overload the value of label, the first value that will be used is the value of label in the descriptor, this represents the *default* value. If the edge has the label field, the descriptor label value will be replaced with the label value in the edge. This rule will be applied to all the forms that are present in the edge branches field. Since this applies to all paths that traverse the edge, there may be cases in which a specific path may need a different label, to handle these cases there is the modifiers field. This is an object where the first level property names must match the elements of the branches field. These properties are objects which contain the fields to be overloaded. If the current branch matches one of the first level properties of the modifiers field, the value should be taken from there. So in our label case, if modifiers contains the current form \_id and it also contains the label field, that value will replace all others. The service that returns form structures takes care of this.

Note that not all fields can be used this way, keys and identifiers, obviously, should not be included in this scheme, the Form.js class in the top level classes directory of the application is used to generate form structures and implements this scheme, there is a static method, modifierFields(), which returns the list of fields that can be included in the edge and the modifiers field.

# Back-end

The back-end is implemented using the ArangoDB Foxx services, this is the structure of the application directory:

APP:

classes:

Annex.js

Descriptor.js

Document.js

Edge.js

EdgeAttribute.js

EdgeBranch.js

Form.js

Identified.js

Identifier.js

Persistent.js

Study.js

Term.js

Toponym.js

Transaction.js

User.js

data:

auth.json

errors.json

messages.json

test.json

dictionary:

Dict.js

handlers:

Annex.js

Descriptor.js

Init.js

Schema.js

Session.js

Study.js

User.js

Utils.js

main.js

manifest.json

middleware:

user.js

models:

annex:

smartRegistration.js

smartRegistrationForm.js

smartUpdateForm.js

descriptor:

descriptorUpdateValidation.js

schema:

schemaEnumList.js

schemaEnumTree.js

schemaGetDocument.js

schemaGetDocuments.js

schemaIsEnumBranch.js

schemaIsEnumChoice.js

schemaTypeList.js

schemaUserSchema.js

schemaValidateForm.js

schemaValidateProperty.js

schemaValidateStructure.js

session:

login.js

managers.js

manages.js

user.js

userProfile.js

userProfileForm.js

study:

studyRegistration.js

studyRegistrationForm.js

studyUpdateForm.js

user:

changeUsername.js

reset.js

signInAdmin.js

signInUser.js

signInUserForm.js

signUpAdminForm.js

signUpUser.js

signUpUserForm.js

utils:

getToken.js

paw:

Services.paw

README.md

routes:

annex.js

descriptor.js

init.js

schema.js

session.js

study.js

test:

Application.js

Descriptor.js

Form.js

Log.js

MyError.js

Schema.js

User.js

Utils.js

Validation.js

user.js

utils.js

scripts:

setup.js

teardown.js

test:

classes:

DocumentUnitTest.js

DocumentUnitTestClass.js

EdgeAttributeUnitTest.js

EdgeAttributeUnitTestClass.js

EdgeBranchUnitTest.js

EdgeBranchUnitTestClass.js

EdgeUnitTest.js

EdgeUnitTestClass.js

PersistentUnitTest.js

PersistentUnitTestClass.js

UnitTest.js

disabled\_test\_Document.js

disabled\_test\_Edge.js

disabled\_test\_EdgeAttribute.js

disabled\_test\_EdgeBranch.js

disabled\_test\_Example.js

disabled\_test\_Identifier.js

disabled\_test\_Persistent.js

disabled\_test\_Transaction.js

parameters:

Document.js

Edge.js

EdgeAttribute.js

EdgeBranch.js

Persistent.js

Transaction.js

test\_Term.js

utils:

Application.js

Constants.js

Dictionary.js

GeoJSONValidation.js

Log.js

MyError.js

Schema.js

Validation.js

# Classes

The easiest way to manage objects in the database is to use, and extend, the set of classes in the top level ‘classes’ directory.



The Document class implements the interface and common methods and is the ancestor of all derived classes. The classes in red correspond to objects stored in the database and implement behaviours specific to their type.

All classes share the same constructor signature and a set of methods that can be used to perform operations:

* constructor(): The constructor instantiates the object and optionally loads it from the database if a document reference was provided.
* setDocumentProperties(): This method can be used to set the object properties.
* validateDocument(): This method can be used to check if the document properties are valid, this method is automatically called before storing the object in the database.
* resolveDocument(): This method can be used to retrieve and load the object from a document stored in the database, using the current object contents as the selector.
* insertDocument(): This method can be used to insert the document in the database.
* replaceDocument(): This method can be used to replace the document in the database.
* removeDocument(): This method can be used to remove the document from the database.

Please look at the test services, unit tests and the study and annex services for examples on how these classes are used. Note that the source code is commented, so you will find explanations there.

# Development plan

This is a tentative development plan in which I set the main components and how these could be developed.

By reading the SMART Survey Aggregator document you will get an idea of what are the main features and activities performed by the application.

## User management

This involves creating, managing and authenticating users. The related services have been implemented:

* session/user: This service returns the currently logged on user.
* session/user/login: This service can be used to log in a user.
* session/user/logout: This service can be used to log out a user.
* session/user/managers: This service can be used to retrieve the hierarchy of users managing the current user.
* session/user/manages: This service returns the hierarchy of users managed by the current user.
* session/user/profile/form: This service returns the form that the user can use to update its personal information, including the password.
* session/user/profile: This service can be used to update the current user’s personal information, including the password.
* user/signup/admin/form: This service returns the form that can be used to register the system administrator.
* user/signin/admin: This service can be used to register the system administrator.
* user/signup/user/form: This service returns the form that can be used to create a user registration.
* user/signup/user: This service can be used to record a user registration.
* user/signin/user/form: This service returns the form that can be used to register a new user.
* user/signin/user: This service registers a new user.
* user/reset: This service can be used to reset a user: it is equivalent to the signup process.
* user/remove: This service can be used to delete a user.
* user/change/code: This service can be used to change a user login code.

Since the services are available, this could be the first component to be developed.

Note that users have a role field which indicates what operations they are allowed to perform, in order to create a user, the creating user, which will become the manager, must have the :role:user value in the role field, if that is not the case, the user should not be allowed to create other users; this is taken care of by the existing services.

## Dataset management

This component’s duty is to register and manage datasets. A dataset is a study that includes the set of documents that were produced by the study.

There are two services that can be used to register a dataset:

* dataset/registration/form: This service returns the form for registering a dataset.
* dataset/registration: This service can be used to register the dataset.

There are other components needed to manage the datasets, this includes listing the datasets, modifying the registration and uploading the annex documents of the study.

All uploaded files should be managed by the front-end, you can create services in the back-end that record references and other data regarding the uploads, but the physical files should reside in a directory.

### Registration

Use the above two services to register the dataset. Once the dataset has been registered, the other operations can be applied to it. These services will write by default to the services collection.

Only certain users should be allowed to register datasets, these should have the :role:meta value in the role user record field. Uploading datasets should only be allowed to users having the :role:upload role.

### Upload annex documents

A study is a metadata record and the collection of original documents produced by the study. A form should be created in which the client will provide the document, its description and type by linking it with the study. The metadata regarding the single files should be recorded in the annexes collection. This metadata should include the file path and all the necessary information to describe what the file is. The physical file should be stored in a directory and the path included in the annex file metadata.

The link between the file and its study can be implemented in two ways: as a field in the annexes record that contains the study \_id or \_key, or as an edge document that relates the study with the annex file. The first solution should be the best one, since this kind of relationship exists only at a single level.

### List and download

Studies are a package that contains the study metadata, the annex physical documents and their related metadata. A study could be downloaded as a zipped package that contains an excel file with one worksheet for the study metadata and a series of other worksheets, one for each annex file metadata; the actual physical files will also be part of the zipped archive. These elements could also be downloaded individually.

To get a list of studies you can follow the same strategy as users: a user has access to all the user records that it has created, along with all the user records of users created by users created by the user (I know, it’s a bit confusing, but you get the idea). The same can be done with studies, since the user who registered the study becomes its manager. This way the view becomes hierarchical and the system administrator is the root user.

### Upload datasets

The core of the application is aggregating similar dataset data, so another function of this component should manage the data. Datasets should be uploaded as CSV files, generated from the datasets belonging to the study. While converting an uploaded dataset file to a data table is possible, this involves supporting a set of data formats that can be a time-consuming task, so the idea is that the user uploads the original dataset as an archive, then he/she extracts a CSV file from its data and uploads it to the application.

When uploading the CSV file there should be a registration form in which the user indicates to which annex file this CSV data belongs to, this allows creating a link between the data and the annex file dataset, and, therefore, to the study.

The service should create a new collection that will live as long as the dataset has not yet been matched with the data dictionary. Once the dataset has been harmonised, the data in the temporary collection will be appended to the final collection, that in the case of SMART surveys would be the smart collection, and the temporary collection can then be deleted.

The harmonisation process should belong to a different component, since this set of operations are mainly dealing with registering resources.

So here is a summary that could be used as a set of menu options:

* Study Management
  + Register Study
  + Upload annex document
  + Upload CSV dataset file
  + List studies

## Dataset harmonisation

Once a CSV file has been registered and linked with a dataset, it is the time to harmonise it, this process essentially follows these steps:

* Identify header and data rows.
* Match columns with data dictionary.
* Correct errors.
* Append dataset in database.

The second and third steps are recursive, this means that users will match the column, consult the errors, correct the values and submit the data again. From the developer’s point of view the easiest thing to do is to repeat the entire process each time there is a correction, from the user’s perspective the easiest thing is to correct the errors and start over from where they were, this means that ideally the application should provide a series of data manipulation functions, since the data is already in the database.

Harmonisation is a process that should only be available to selected users, these should have the :role:clean value in their role field.

### Identify header and data rows

This step involves displaying the data table to the user and prompting the user to select the *header row* and the first *data row*. The header row represents the row that contains the *variable names*, the first data row should be the first row in which the *data records* can be found; we assume that if that is row 3, rows 3 to the end of the table are the data.

In this step the user should also be given the opportunity to delete columns that are not of interest, so that the table only contains data relevant to the application.

Once this has been done, the application should first check if the header row contains duplicate values. We assume the header row contains the variable names, so these values must be unique: if there are duplicates, the user should be prompted to change their values.

Once the header row contains unique values, the temporary collection can be processed so that each record is a document where the property is the variable name and the value is the corresponding variable value. It is important to filter these properties by removing any null or empty value: the database assumes a value either to be there or not, empty values are not permitted.

### Match columns to data dictionary

This process should be repeated for each column in the dataset. It involves matching each column to a specific descriptor of the data dictionary.

The application could do a preliminary selection of possible candidates by matching the header values with the sym field, synonyms, in the descriptors collection. This field is an array containing all known *variable name synonyms* for a specific descriptor. This list could also be updated as new datasets are harmonised, so that the more you use the application the more it can help you.

Matching could be done using two window panes: on the left you have the dataset table and on the right you can have the data dictionary. The user will select a column in the left pane and then select the corresponding data dictionary entry in the right pane. Data dictionary elements selected using the synonyms could be placed at the beginning, so that they are evidenced, the rest of the data dictionary could be presented as a hierarchy similar to a directory structure.

Once the user matches both the column and the descriptor, the application should register that match and start validating the data in the matched column according to the constraints of the descriptor. This process will generate a series of errors that should be recorded and made available to users so that they can correct the errors.

The core of the application is the harmonisation process, as stated before, repeating the whole process for each set of errors make the life of developers easier, but being able to record the current state make the life of users easier. The application should provide a set of data manipulation functions that can be applied to the data in the temporary collection, preventing the user from having to download the data, correct it and start the process again. These are some of the functions that could be implemented:

* Replace value. When correcting controlled vocabularies you may often encounter misspelled codes, or codes that belong to a different standard, in that case the user should be allowed to replace one value with another for the whole column.
* Add controlled vocabulary element. The controlled vocabulary in the data dictionary might not be complete and some elements might need to be added: giving this possibility would allow new needed elements to be included in the standards.
* Unit conversion. Descriptors have two fields that deal with units: unit and units. The first field indicates what unit the values should be in, for instance a length descriptor might be defined to hold centimetres. The second field is an array that contains the possible units in which values may be found, in the current example it could contain millimetres. It would be useful to create a set of unit conversion scripts so that if the user discovers that the data was provided in millimetres, it could be possible to make the conversion explicitly.

These are just a couple of examples to illustrate how the application could be refined in time.

### Correct errors

If errors need to be corrected, the user should be able to download the current version of the data in the temporary collection in CSV format and make the corrections using a familiar tool such as Excel. Once the user has corrected the errors he/she will need to upload the data again to continue the harmonisation process. This step should not start from the beginning again, the matches already performed should have been recorded and applied again when the CSV file is uploaded again. This means that the harmonisation process should record the current state of the dataset, so that users can continue working at the point they stopped.

### Add new data dictionary entries

Sometimes users may encounter variables of interest which are not yet in the data dictionary, there should be the ability for users to add new elements to the data dictionary. The new elements could be new descriptors and new controlled vocabularies. This last type of elements will be probably the most used feature, especially when handling administrative units which tend to change often. Adding a new element is done in two steps:

* *Suggest change*. The user, which must have the :role:suggest role, creates the new controlled vocabulary element or descriptor. This descriptor will have its status field set to :state:status:proposed, which means that the element exists, but is not yet endorsed. This means that it may be used to match dataset variables, but it should be clearly visible that it is not yet an “official” element of the data dictionary.
* *Accept change*. There is a set of users that have the duty of curating the data dictionary, these users should have the :role:dict role, they will periodically revise the suggested data dictionary elements, make eventual modifications or additions, and finally mark these elements as “official”, by removing the :state:status:proposed status.

### Append dataset

Once all the columns have been harmonised and the data contains no errors, the data can be appended to the final collection.

A set of fields will have to be added to each record in order to make the link between the study and the data:

* Dataset reference. Each record should contain a reference to the dataset to which it belongs, this will allow the record to be also linked to the study that contains the dataset.
* Descriptor usage. The study dataset metadata should be updated to contain the list of used descriptors and their frequency, this allows selecting both data and studies based on what type of information they contain.
* Geographic or administrative unit coverage. Most datasets will have a field that indicates where the observation was done, but if a study was done in a specific location it might be possible that this location is not included in the record, so this information should be added at the record level. If the information is available at the record level, a field should be added to the dataset which collects the distinct locations present in all records, this information can then be used to search datasets and studies by geographic location.
* Date. As with geographic coverage, time tracking is important. In general the date of the observation will be part of the individual records, but if that is not the case it should be added from the information of the study, in each individual record. If the information is available at the record level, a field should be added to the dataset which collects the range of dates present in the dataset record, this will allow searching datasets by date.
* Statistics. Quantitative fields of the dataset records should be summarised for range, standard deviation and other statistical features. In SMART surveys this information is available in the report and the values could be copied to the dataset metadata when registering it, in other cases this information should be computed and added to the dataset metadata record.
* Identification. Each record should have a unique identifier, the easiest way would be to let the database assign the \_key and add a line number field to each record. This field would hold the line number of the original dataset file from which the record comes, the combination of the dataset reference and line number will allow selecting the original record in the dataset file, while the database-assigned \_key will ensure all records are unique.

### Cleanup

Once the temporary collection contents have been appended to the final collection, the temporary collection can be deleted and the dataset marked as harmonised.

Only users having the :role:commit role should be allowed to perform this operation.

## Queries

The final goal of the application is to provide access to data aggregated by dataset type. This should be done by providing a query interface to the user. Since the list of available data variables is the list of descriptors defined in the database, the first component of the query interface should be a way of navigating the data dictionary.

As described earlier, the list of *used* variables can be found in the dataset metadata, this is important, because it allows us to immediately restrict available variables only to those that have associated data. By making a distinct value query on the list of variables belonging to the study datasets we can have the list of variables that can be proposed to the querying user.

The idea is to create an *ontology* of variables, grouped by scope or category, so that the user is guided in the search of which variables to select. The distinct list of variables from the datasets metadata will allow us to filter those variables that do not have any associated data, so that users will have a meaningful result.

Query conditions depend on the variable data type, strings could have a *starts-with*, *contains*, etc. while numeric values could be selected with ranges. A further top level condition should be added which requests the *presence* of the variable: this means that we are looking for all resources that *feature* that variable.

The workflow could be:

1. Select the desired variable.
2. Enable a checkbox or button that indicates we want everything that *contains* that variable.
3. If the above button is set, disclose a pane that allows users to add search conditions on the variable’s values.

Once a query clause is compiled, the database could be searched as follows:

1. Select all data that satisfies the query.
2. Select the distinct list of datasets from the data selection.

Although this query could be done in one AQL command, this operation should be divided into two distinct phases:

1. Select all studies that satisfy the query conditions.
2. Select all data that satisfies the query conditions.

The first query should simply select all datasets that have the requested variables, in the second query the actual query conditions should be applied[[14]](#footnote-14). The results should be presented in two distinct panes, one will contain the list of studies and the other will contain the list of data. Users can then select the studies and decide whether to download the full archived study, or download individual annex files of the study, such as reports and datasets. User can also request a download of the raw data displayed in the other pane as a CSV file.

In order to perform a query, the user should have the :role:query role and, if the user also has the :role:download role, he/she can download the results of the query, besides consulting them.

Study metadata contains mainly summary data, there should be a mechanism to identify which summary variables can accept query conditions. For instance, suppose we have a dataset that records the height of 5 years old children: the raw data records will contain the height of the specific child, while the study dataset should record the range of heights found in the dataset. A request for height within a specific range can then be applied to both the dataset summary data and the raw record data. This, however, poses a problem: the raw data variable is a number, while the summary data variable must be a range, this means that there must be a link between those variables and this has to be implemented.

1. The idea is that each data domain represents a compatible set of data. While the use of a NoSQL database would allow you to store all types of data into a single collection, when the number of records grows, it is advisable to aggregate similar data structures together to allow specific indexing strategies. [↑](#footnote-ref-1)
2. The process of harmonizing data with the data dictionary has been described extensively in the *SMART Survey Aggregator* background document. [↑](#footnote-ref-2)
3. The management of study annex documents should be performed by the front-end component, ArangoDB does not support a viable binary format that would allow to store physical binary files as records – as MongoDB [GridFS](https://docs.mongodb.com/manual/core/gridfs/) allows. [↑](#footnote-ref-3)
4. ArangoDB has two types of collection: document and edge. This collection is one of the edge collections used in the database, the name may be misleading, this collection only contains edge documents related to toponyms. [↑](#footnote-ref-4)
5. There is only one record in the database that violates this rule: it is the default namespace, this namespace is reserved for default terms and descriptors, the exception is used so that it is not possible to use that namespace in other types of terms and descriptors. [↑](#footnote-ref-5)
6. ArangoDB does not make a distinction between integer and real numbers: everything is a floating point number, as in JavaScript. If a value must be an integer, there is a specific data type for it. [↑](#footnote-ref-6)
7. Only relevant for non-integers. [↑](#footnote-ref-7)
8. The key data type can be any data type derived from the base text data type, it is useful when defining an object whose field names should be taken from a controlled vocabulary. [↑](#footnote-ref-8)
9. Note that there is another reference data type that defines fields that point to the \_id of another object: this type does not derive from string since it adds the collection name in the value which may exceed the string maximum character length. [↑](#footnote-ref-9)
10. This means that the referenced term must exist in branched edge branches field. [↑](#footnote-ref-10)
11. There are many of these procedures that are declared, but do nothing yet, only the essential ones have been implemented: the controlled vocabulary is there as a list of possible options. [↑](#footnote-ref-11)
12. In a previous prototype of this system the \_key of the descriptor was the hexadecimal value of a counter, while the gid represented the “human-readable” representation of the identifier. This allowed to have a large number of descriptors and keep the field name short at the same time. In this version we did not implement this feature, which means that the choice of the \_key is the responsibility of the user defining the data dictionary. [↑](#footnote-ref-12)
13. This reference type exists only because with descriptors the \_key represents a database field name which could be different from the descriptor global identifier. Normally descriptors would have the same structure and identification scheme as terms. [↑](#footnote-ref-13)
14. This is because at the study level we only have summary data. [↑](#footnote-ref-14)