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.
* **\_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.
* **\_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.
* **\_to:** This field represents the relationship destination node \_id.

## 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 connot 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. *Note the parenthesis in the data type key: I add them to separate the code from the colons used in the text.*

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

Data types are organised in a hierarchical way, meaning that a generic domain data type may be the parent of a more specific data type. For instance, the text data type defines text; the string data type derives from the text data type, but adds a size limit constraint. The base data type determines which modifiers derived types can apply. For instance the text data type family can have a minimum and maximum number of characters; the number data type family can have a minimum and maximum value range.

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: length, 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:value:set**]: This data type represents a *list* or *array* of *unique* items of undefined type. This base type derives from the list type, it only adds the constraint that elements should be unique.
* [**: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 respectively indicate the data type of the pair key[[8]](#footnote-8) and value.
* [**:type:value:ref**]: This data type represents the parent of all text types that represent a reference to another record in the database. You should consider this type as virtual, since only types derived from this one can be used[[9]](#footnote-9).

### Text derived types



* [**: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 yext type and represents a string that is limited to 254 characters. This type is used for all text variables that can be indexed and used as object identifiers.

### String derived types



* [**:type:value:key**]: This data type is used for strings that are used as record keys, it uses a regular expression to restrict the allowed characters to those compatible with ArangoDB indexes and keys.

## 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[[10]](#footnote-10).

* **\_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 to the term or descriptor that represents the variable’s *namespace*.
* **lid:** This field contains the local identifier unique within its namespace (nid).

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. There are three types of concrete reference types: :type:valueref:id, :type:value:ref:key and :type:value:ref:gid, they represent respectively a reference to a record \_id, \_key and gid; the last two data types have a modifier that indicates to which collection the record must belong. [↑](#footnote-ref-9)
10. 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-10)