# RIS3D, a Referenced Information System in 3D Supplemental materials

BRUNO DUTAILLY, Archeovision UMS3657, Centre National de la Recherche Scientifique, France
JEAN-CHRISTOPHE PORTAIS, Ministère de la culture, Direction Régionale des Affaires Culturelles Nouvelle-Aquitaine, France
XAVIER GRANIER, Institut d'Optique Graduate School/LP2N, France

**ACM Reference Format:**
Bruno Dutailly, Jean-Christophe Portais, and Xavier Granier. 2022. RIS3D, a Referenced Information System in 3D Supplemental materials. *ACM J. Comput. Cult. Herit.* 1, 1, Article 1 (January 2022), 6 pages. https://doi.org/10.1145/3517043

In the following supplemental document, we are providing the readers with even more detailed choices on our implementation. This concerns types, structure, non 3D features, query in JSON, converters, and user interface for data manipulation regarding types...

## 1 TYPES

The table 1 shows a list of those types, and how they are handled in our implementation.

Table 1. Type handling in RIS3D

| Type of data | Type name in RIS3D | Storage in JSON | PostgreSQL query |
|---|---|---|---|
| **Integer and floating point number** | int, double | int, double | Comparison functions are available: equal, superior, inferior, simple operations |
| **Free text** | string | string | Comparison functions are available: "like", "contains", ... |
| **Boolean** | bool | bool | Test if true of false directly |
| **List of words or texts** | enum("text 1","text 2",...) | string | Same as string type |
| **Date, time, and datetime** | timestamp | timestamp | Many functions for comparison, operation and detail extraction |
| **Image** | image | string: path to the file | Test if filled |
| **Any document** | file | string: path to the file | Test if filled |
| **Color** | color | string: text representing the color as hexadecimal RGBA | Text comparison, no specific function provided |

Authors' addresses: Bruno Dutailly, bruno.dutailly@u-bordeaux.fr, Archeovision UMS3657, Centre National de la Recherche Scientifique, Archéopôle, Esplanade des Antilles, Pessac, France, 33600; Jean-Christophe Portais, jean-christophe.portais@culture.gouv.fr, Ministère de la culture, Direction Régionale des Affaires Culturelles Nouvelle-Aquitaine, 54, rue Magendie, Bordeaux, France, 33074; Xavier Granier, xavier.granier@institutoptique.fr, Institut d'Optique Graduate School/LP2N, LP2N-IOA, rue François Mitterand, Talence, France, 33400.


XXXX-XXXX/2022/1-ART1 $15.00
https://doi.org/10.1145/3517043

## 2 STRUCTURE

Figure 1 shows an example of a typed structure and its JSON definition. All values are strings and contains the RIS3D type of each field. The figure 2 shows two records conform to the structure. The first fit fully the structure whereas the second is incomplete but still correct.

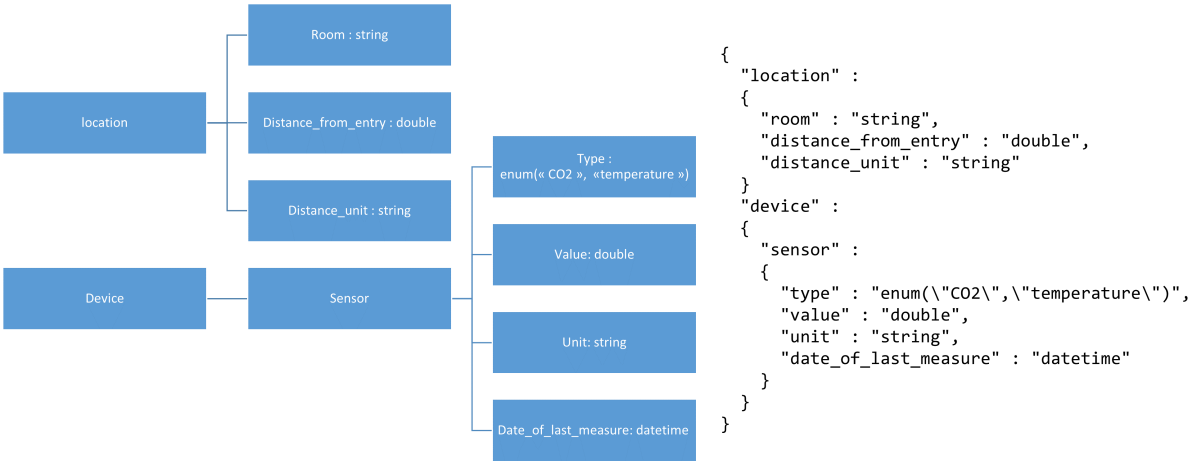


Fig. 1. Example of typed structure for storing user's data and its JSON representation.

## 3 NON-3D FEATURES

Implemented non 3D features in our RIS3D are:

- User permissions: user accounts with group management and access control list to data.
- Type structure: an interface to create and edit the structure and type of data through a WYSIWYG JSON editor.
- Export: the database can be exported to JSON, CSV or SQL formats.
- Import: data can be imported as CSV format. It consists in defining a mapping between columns of the CSV and field in the data tree. This mapping is done through an interactive web application using pure javascript and JQueryUI. Once this mapping done, the CSV is parsed and feed the database with some data conversion if needed and a row range selection.
- Raw view: list all records of the database with the ability to delete each record one by one.
- Layer view: list all layers with the ability to export all records answered by the query to CSV and XLS formats.
- Update and fixes: finds errors in the database structure and provides actions to fix them. This module is capable of create fields in the database and change types if needed.
- Inconsistency: list issues in database records like an address in the data tree that is not defined in the type structure, or a text in a record typed as "list of texts", but not present in the list, etc.

## 4 EXAMPLES OF WEB SERVICE COMMANDS AND QUERIES

As examples, we can cite, with the format **"action"**(keys):

- **"get_me"**: return all data belonging to the logged user including layers, ACL, personal information…;
- **"get_shared_layers"**: return all layers shared by users in the same groups of the logged user;

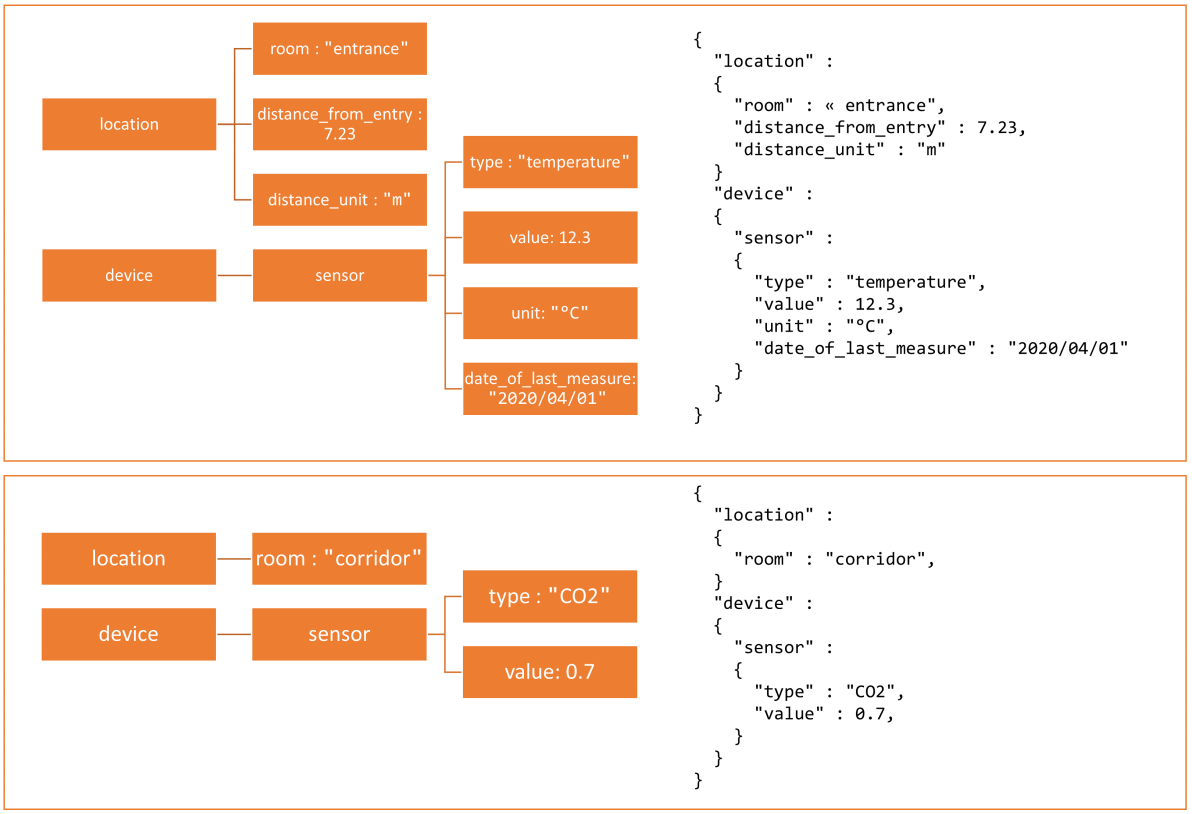


Fig. 2. Two examples of user's data fitting the structure definition and their JSON representation.

- **"get_structure"**: return the typed structure tree;
- **"set_record_json"**(table,path,id,value): set the `value` at `path` of record `id` in `table`;
- **"add_point"**(x,y,z,nx,ny,nz,data): add a new 3D point `(x,y,z)` with normal `(nx,ny,nz)` in table point with `data` as JSON;
- **"query"**(select,from,where): query the database and return records.

Since queries must not be related to the database engine, we define a query structure as a JSON object. The web server will translate it into the right query language, ensuring our extended types to JSON standard types will be nicely handled. The query structure is made up of three keys:

- **"select"**: an array of table fields to get in each record. If only one field is needed, the expected array can be a string;
- **"from"**: an array of tables names. Tables used by `where` will be automatically added to the from table list;
- **"where"**: a sub-object containing conditions and operators. A condition is a path in the user's data tree to a chosen field, a boolean-valued function, and an optional argument. All conditions can be organized with logical operators, basically "AND" and "OR" operators. This organization supports a construction in a tree to allow priorities in the evaluation of these conditions. Figure 3 shows an example of `where` statement of a query to match records between two dates and containing one of two keywords.

In the web server, a recursive evaluation ftattens the JSON object into the database language (cf. Figure 3).

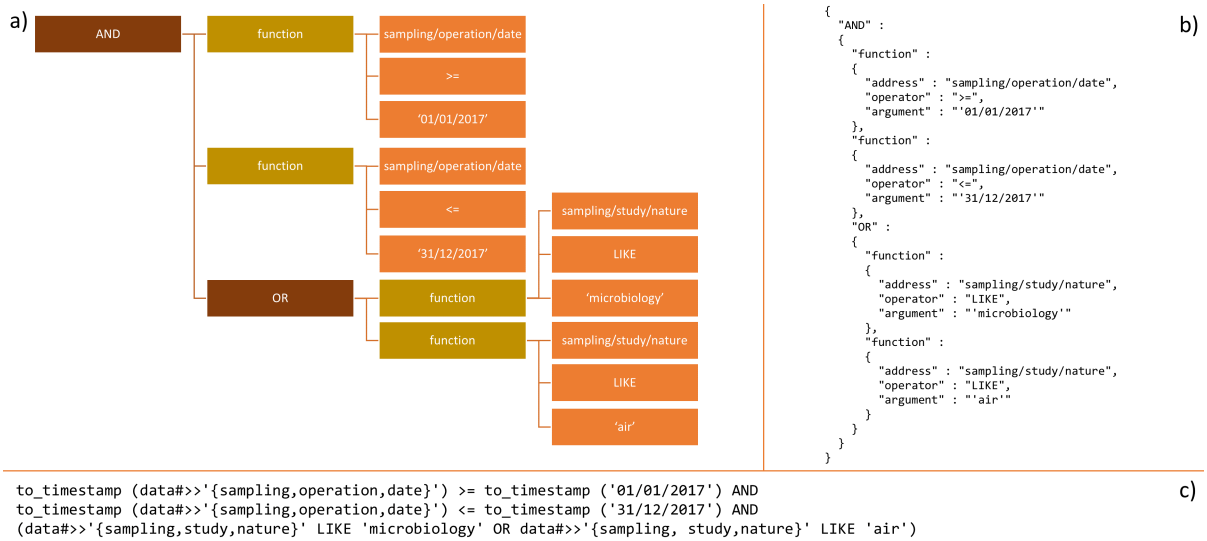


Fig. 3. where statement of a query as a tree representation (a), written in JSON (b) and translated to PostgreSQL language (c).

The following example shows a query in JSON format and how it is flattened to a SQL query for PostgreSQL.

```
{
 "AND" :
 {
  "function" :
  {
   "address" : "location/room",
   "operator" : "==",
   "argument" : "'entrance'"
  },
  "function" :
  {
   "address" : "device/senor/temperature/value",
   "operator" : ">",
   "argument" : "12"
  }
 }
}
```

will result in PostgreSQL language and a JSONb field named "data" to:

```
data#>>'{location,room}' = 'entrance' AND data#>>'{device,sensor,temperature,value}' > 12
```

Types not supported natively by the database engine and custom operator not supported too, must be translated to a valid SQL query. This is done on the server side since the user must not suffer from database engine limitation. The following example shows how custom operators on custom types are translated:

```
AND
address : "device/sensor/temperature/photo/image", operator : "EXISTS", argument : null
```

`address : "device/sensor/temperature/photo/date", operator : ">=", argument : "2020/04/01"`
will result in PostgreSQL language to:

```
(data#>>'{device,senor,temperature,photo}')::jsonb ? 'image' AND
to_timestamp(data#>>'{device,senor,temperature,photo,date}') >= to_timestamp('2020/04/01')
```

The operator "EXISTS" is transformed to the "?" operator on JSON field, and the type "date" given as a text is converted into a postgres timestamp.

## 5 CONVERTERS

Table 2 shows a non exhaustive list of converters. Implementation of those converters relies on technical choices. This list must grows regarding the needs of users.

Table 2. Type converters. * are converters with arguments

| From ↓ to → | Integer | Floating point | Text | Date | Boolean | Color | List of texts |
|---|---|---|---|---|---|---|---|
| **Integer** | | to double | to text | timestamp to date | to bool | index and RGBA to color | |
| **Floating point** | round, floor, ceil | formula* | to text | | is positive | | |
| **Text** | char count, parse int, index of* | parse double | concat text* | parse date* | is empty | Hex to color, name to color | split* |
| **Date** | to timestamp, year, month, ... | | to ISO, to format* | add days*, add minutes*, ... | is before* | | |
| **Boolean** | to int | | to text* | | not | to color* | |
| **Color** | to RGBA | | to name | | | set alpha*, keep red, ... | |
| **List of texts** | list count, index in list | | to text | | is in list | to color | index prefix |

## 6 VISUALISATION AND EDITION OF DATA REGARDING THEIR TYPES

The table 3 shows how data is displayed and edited regarding types.

Table 3. Data visualization and editing in RIS3D

| Type of data | Visualization | Edition |
|---|---|---|
| **Integer and floating point number** | as text | Input text field |
| **Free text** | directly in UI or 2D sprite | Input text field |
| **Boolean** | as text | check box |
| **List of words or texts** | as text | Input text field with completion and/or validation and combo box |
| **Date, time, and datetime** | as text in several format | interactive calendar widget |
| **Image** | as an image in UI or 2D sprite | Dialog box to browse to the image |
| **Any document** | as an icon in the UI, opened by the default application on the computer host | Dialog box to browse to the file |
| **Color** | as a colored item in UI or in 3D | Color picker dialog box |