Towards rich and standardized metadata in iRODS

Mariana Montes
KU Leuven
W. De Croylaan 52, 3001 Leuven,
Belgium
mariana.montes@kuleuven.be

Paul Borgermans
KU Leuven
W. De Croylaan 52, 3001 Leuven,
Belgium
paul.borgermans@kuleuven.be

ABSTRACT

Metadata is a crucial feature to manage and find data in iRODS, especially if used in a systematic way. However, human manipulation of metadata is prone to errors, from typos to inconsistency in case, spelling and format. In order to tackle this issue, we have developed a "metadata schema" management tool in which an iRODS user can design a form meant for systematic application of a specific metadata schema. This form consists of a collection of fields of different types: from different scalar input fields through multiple-choice fields to composite fields. When a user adds metadata using this schema, they get a form that includes validation, which can relate to the format or the possible values. The iRODS attribute name of the metadata inserted via a schema follows a pattern with namespacing, including the identifier of the schema as a prefix. In addition, namespacing can be used to generate nested fields and thus render hierarchical metadata structures.

The metadata schema itself is stored in JSON format, which can also be used to import and export its contents. Moreover, a life cycle was designed so that only stable schemas can be used for metadata annotation but at the same time the schema can evolve into new versions. Concretely, it is possible to have multiple versions of a schema, among which: a draft that can be edited, a "published" version to be used in annotation, and all "archived" versions, which cannot be used anymore.

Via this tool, we expect users to be able to design metadata schemas of varying complexity for their whole team to apply systematically. This will increase the uniformity and thus usability of metadata and can be used to enforce the inclusion of metadata to certain collections or data objects.

Keywords

metadata, standardization, web-interface, javascript

INTRODUCTION

Metadata is key for the description and contextualization of research data. However, its reliability strongly depends on its systematicity: when searching for data via metadata, a user needs to know how that data was annotated in the first place, e.g. what metadata tags to expect, what kinds of values are used, in which language, etc. This also means that when applying metadata, a user must be mindful of how it will be used, what can be expected. Particularly when data requires a richer, more complex description, the possible ways of coding it with metadata grow exponentially, making it increasingly harder to maintain.

Metadata schemas offer a solution to these challenges. In this paper, we introduce a comprehensive feature of the ManGO portal [1] that allows users to create, manage and apply metadata schemas. Concretely, metadata schemas address the following three issues:

- Assignment of the appropriate metadata following a standard, via web forms and input validation.

iRODS UGM 2023 June 13-16, 2023, Chapel Hill, NC
Authors retain copyright.

- Hierarchical metadata, via namespacing and a custom use of units.
- Flexibility and consistency across time via versioning and a life cycle.

The ManGO portal offers two linked features to deal with metadata schemas. On the one hand, the metadata schema manager allows users to create, edit and manage metadata schemas across their full life cycle. On the other hand, it is possible to assign metadata to a data object or collection based on the published version of a metadata schema. (These concepts will become more clear in the following pages.)

In the following section the namespacing will be described, focusing on how it addresses the main challenges mentioned before. Afterwards we will present the metadata schema life cycle, i.e. how we ensure the stability of the metadata schemas while allowing for adjustments and developments. The third section takes a more technical perspective, introducing the JSON format used to register the metadata schemas.[1]

NAMESPACING IN MANGO METADATA SCHEMAS

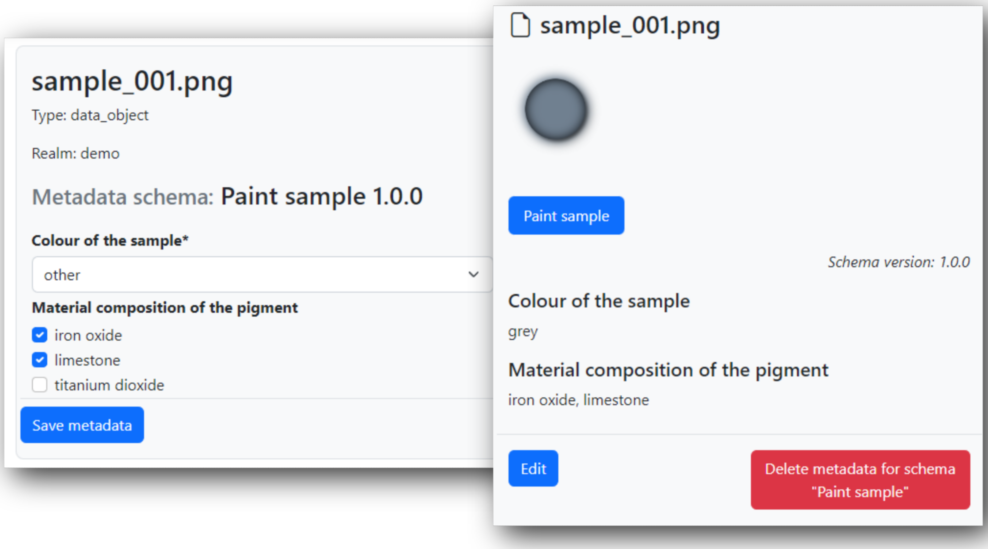
In the broadest sense of the term, metadata is data about data, i.e. description of data that can be embedded inside a file, attached to it as a tag or contained in an additional document (*aka* sidecar file). In iRODS, it takes the form of AVUs (attribute-value-unit tuples) stored in the iCAT database [2]. The flexibility of the system gives users a lot of freedom on the kind of metadata they can apply, leaving open the issue of standardization.

As an example, let's suppose that we store pictures of paint samples and we want to indicate the colour of the sample as metadata. For a given grey sample, we could create an AVU with "colour" as the attribute name and "grey" as the value, leaving the unit empty, i.e. `(colour, grey,)`[2]. But a colleague that uses a different variety of English may use (or expect) a different spelling, i.e. `(color, gray,)`, or even `(COLOR, Gray,)`. It could even be the case that in our plurilingual team someone uses `(color,gris,)` and someone else `(kleur, grijs, )`. Even within the same language, different levels of specificity may be valid to describe the same item, e.g. `(colour, slate gray,)`. We might want also want to reduce variation by picking a standard and thus use a HEX value, generating an AVU such as `(colour, #708090,)`, or even `(colour, #708090, hex)`. In short, even a very simple description of an item can be executed in a myriad of ways: proper standards must be established *and enforced* to keep the metadata informative and useful.

In order to tackle this challenge, we can use **metadata schemas**, i.e. sets of instructions to enforce standards. In practice, instead of manually and freely inputting an attribute name and value, a user would be able to fill in a form, which provides two great advantages: the AVU name is unique and immutable and the value can be validated.

The uniqueness and immutability of the name is ensured when the metadata schema is designed, via namespacing. First, the metadata schema itself has a name that is unique within the realm (e.g. a user, a collection, a zone...) in which it is used, e.g. 'paint_sample'. Second, the **field** corresponding to a given AVU name also has a name that is unique within the schema, e.g. 'colour'. When actually filling an AVU linked to this field, its name will be fixed to a combination of these names, with an additional prefix, e.g. 'mgs'[3]; this prefix is meant to identify ManGO-specific metadata, as opposed to free metadata or metadata from other sources. Concretely, AVUs linked to this field will have as name 'mgs.paint_sample.colour', which is then sure to be unique in its realm and brings the AVU together with other AVUs originated in the same schema, e.g. 'mgs.paint_sample.composition'. Once the schema has been designed, a user applying metadata based on the schema cannot modify the name anymore.

[1]Figures in this paper are screenshots of the application; the example used is available demo in a public repository (`https://github.com/kuleuven/mango-metadata-schemas`).
[2]Across the paper we will represent AVUs in parentheses, with the attribute name, value and optional unit separated by commas.
[3]In ManGO this prefix is 'mgs', which stands for 'ManGO schema'; a different implementation use a different prefix.



sample_001.png

Type: data_object

Realm: demo

Metadata schema: Paint sample 1.0.0

Colour of the sample*: other

Material composition of the pigment:
- [x] iron oxide
- [x] limestone
- [ ] titanium dioxide

sample_001.png

Schema version: 1.0.0

Colour of the sample: grey

Material composition of the pigment: iron oxide, limestone

Figure 1. Form to apply metadata from a schema and view of metadata after application.

The validation of the values occurs via form validation on the client side of the web-application[4]. When designing a metadata schema, a user can choose among a variety of input types: from basic text input, numeric input and dates or datetimes, to dropdowns, radio buttons and checkboxes. Text input can be limited to email addresses or URLs and/or checked against any custom regular expression; numeric inputs can have limiting ranges and distinguish between integers and floats. Dropdowns, radio buttons and checkboxes allow the user to define a limited set of possible values, reducing any variation due to typing. Given our example with `(colour, gray,)`, we could create a dropdown with possible color names; if we prefer to use hex codes, we could generate a text input restricted by the regular expression `/^#\d{6}$/`.

Note that this format still allows users to create multiple AVUs with the same name for a given item. One way is by creating a checkbox or a multiple dropdown, in which a user can choose more than one value. Another option, available for the simple inputs, is to make the input **repeatable**: when a user fills in the form to apply metadata, they will be able to duplicate the field by clicking on a button, and thus generate as many instances as they want.

Next to the immutable names and the validation, a final advantage of using a form to input metadata linked to a schema is that the user can see which fields are expected, among which some may be required and others, optional. In the ManGO portal, when we view the metadata linked to a schema we also get to see all the possible fields and for which there is metadata or not. This is particularly useful as schemas grow in complexity, and not all users may remember which are all the metadata fields that are necessary or at least recommended (see Figure 1).

To sum up, metadata schemas, in its most basic form, provide the following features to facilitate rich, standardized metadata:

- Names of AVUs are unique, immutable and conventionalized.
- Values of AVUs are validated based on custom rules.

[4]But see `https://github.com/kuleuven/mango-schema-validator` for a Python package to validate a dictionary of metadata values against a metadata schema.

- Metadata linked to a schema is conceptually unified by the namespacing (sharing the name of the schema); accessing a schema provides a standardized list of recommended and compulsory fields.

However, metadata schemas offer more: thanks to namespacing, metadata can be organized hierarchically.

Hierarchies

Metadata schemas are not just sets of instructions to standardize metadata, but constitute coherent collections of properties. For example, in the case of the paint sample, the schema could include various physical properties such as colour, material composition, etc. We might also want to register information about the source of the sample, e.g. a certain painting, which has properties such as title, year of creation, storage location... These are properties of the painting, which is itself a property (source) of the paint sample. How can we include this in the metadata schema without flattening this hierarchical structure? And what if one of the properties of the painting has the same name as a property of the paint sample?

In such a situation, we can solve the problem with namespacing. Within ManGO, complex properties such as 'source' in this example are called **composite fields** and are treated as miniature schemas included inside the main schemas. When designing a metadata schema, a user can choose this type of field as an alternative to **simple fields** (text boxes, numeric input, dates...), **single-value multiple-choice fields** (normal dropdowns and radio buttons) and **multiple-value multiple-choice fields** (multiple dropdowns and checkboxes). When choosing a composite field, they can assign it a name (e.g. 'source') and then add fields to it, such as a normal text input 'title', an integer input 'year', a dropdown with 'location' options, etc. But even though the composite field is created as another field at the same level of, say, 'mgs.paint_sample.colour', there will not be AVUs with the name 'mgs.paint_sample.source'. Instead, the AVUs will correspond to the fields inside the composite field, which will include the name of the composite field in the namespacing chain: 'mgs.paint_sample.source.title', 'mgs.paint_sample.source.year', etc.

This solution provides a number of benefits. In the first place, the longer chain of names makes the hierarchical structure explicit, even though the storage of the AVUs themselves in the iCAT database is not hierarchical. Therefore, even if the attributes of the painting are brought together with the attributes of the paint sample in the same larger schema, it is clear that they are not at the same level, i.e. they do not describe the same thing. Second, the name of the composite field itself is also meaningful and unique, indicating what property of the item they describe (the "source"), while linking it to its own properties. This way, an AVU named 'mgs.paint_sample.id' represents an identifier of a paint sample, whereas 'mgs.paint_sample.source.id' represents an identifier of a *source of* a paint sample.

Like simple fields, composite fields can be repeatable. For example, let's suppose that we don't want to just add the 'composition' metadata as a string of materials (slate, iron, lime...), but we would like to match a percentage to each material. In that case, we could create a 'composition' composite field with two simple fields inside: 'material' and 'percentage'. By repeating the composite field, we can insert multiple materials with their respective percentages. In the visual rendering of the form and the metadata in the ManGO portal, material and percentage are visually paired via boxes and dividers. In iRODS itself, the pairing is indicated via units. For example, if we indicate that the pigments of the paint are made with 70% lime and 30% iron, we would obtain the following AVUs: `('mgs.paint_sample.composition.material', 'lime', '1')`, `('mgs.paint_sample.composition.percentage', '70', '1')`, `('mgs.paint_sample.composition.material', 'iron', '2')` and `('mgs.paint_sample.composition.percentage', '30', '2')`. The unit '1' brings the first pair of material and percentage together, and the unit '2' the second pair.

LIFE CYCLE

One of the main goals of metadata schemas is to support standardization and conventionalization of metadata. However, as a research project advances —and as researchers get more familiar with their data, how they use it and how best to describe it— these standards and conventions may need to adjust. For example, we might have started the 'paint_sample' schema with a checkbox for the 'composition' field, and after a few weeks of applying the metadata, we realize that it would be useful to add information on the percentage of each material. The solution is to create a repeatable composite field, as described above, but how does such a change in the schema affect the metadata that has already been applied?

One option would be to just modify the schema: now the ‘composition’ field is composite instead of multiple-value multiple-choice, and the name of the material is coded as ‘mgs.paint_sample.composition.material’ instead of ‘mgs.paint_sample.composition’. For any new item, the metadata will follow this new setting, but older items will still have ‘mgs.paint_sample.composition’. This can become quite confusing. More importantly, the standard embodied by the schema is not so much of a standard if it can suddenly, quietly change in this way.

Another option would be to make schemas immutable and instead create a new schema, with a different unique name (e.g. ‘paint_sample2’), that includes the composite field ‘composition’ instead of the multiple-value multiple-choice version. In this way a user knows, to a degree, that the data annotated with the ‘paint_sample’ schema followed different instructions from the one annotated with the ‘paint_sample2’ schema. However, it is not so straightforward to realize that ‘mgs.paint_sample.colour’ is actually equivalent to ‘mgs.paint_sample2.colour’; there are no restrictions on how to name this new “version” of the schema, and we could even apply metadata to the same item based on both schemas. In addition, this would generate an increasing number of schemas that may become unmanageable.

Instead, the ManGO portal supports a schema life cycle: multiple **versions** of a schema can coexist, of which at most one is **published** and at most one is a **draft**. The version number is included as another AVU ‘__version__’ with the same namespacing as a top level field, e.g. `('mgs.paint_sample.__version__', '1.0.0',)`, and shown when applying and viewing metadata linked to a schema. Crucially, only one version at a time can be edited (the draft) and only one version at a time can be used for applying metadata (the published version).

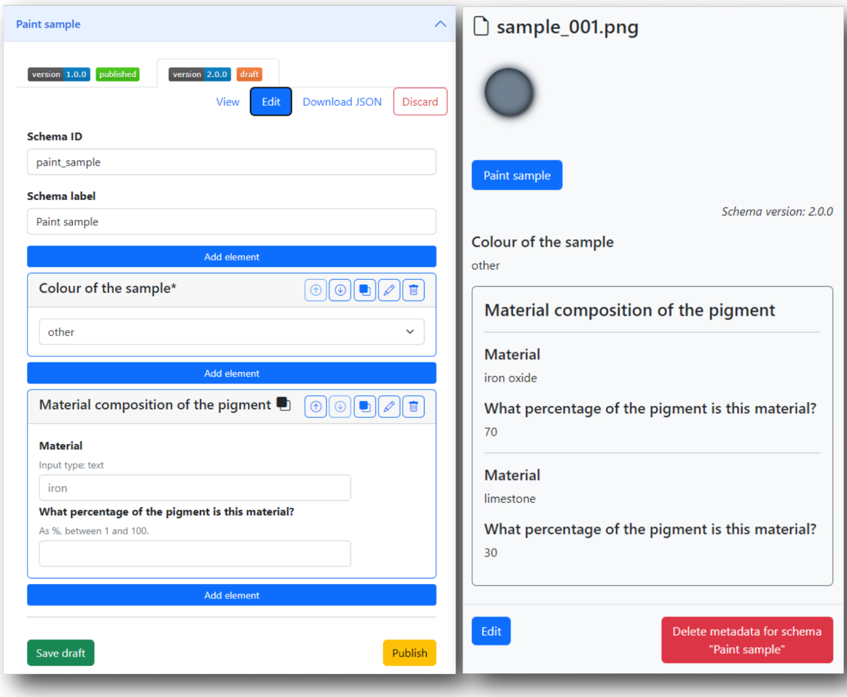
Concretely, when a metadata schema is first created, it is mostly saved as a draft with version 1.0.0. This draft can be edited and even deleted, but it is not available for applying metadata, since it is unstable and thus unreliable. In order to use the schema for applying metadata, it must be published. This changes the status of the schema so that it cannot be edited or deleted any more: it is formally registered and preserved, both as instructions to apply metadata and as documentation of the standard itself. If a user wishes to edit the schema, they can create a new draft, with version 2.0.0[5]. This draft can also be edited and deleted but not used. If a user wants to delete a published schema it will be **archived**: it will still exist, since it documents how data was annotated based on it, but it cannot be edited nor used. It also records the history of the schema. Once the new draft is ready to be used, it can be published, which automatically archives any currently published version. Any new metadata will be based on version 2.0.0; metadata that followed the standard of version 1.0.0 will be marked as such by the ‘__version__’ metadata of the item.

If a user tries to edit the metadata based on the schema after a version change, they will be provided with the latest version of the form: compatible fields will be kept, missing fields will be deleted and new fields will be added. For example, say that we annotated a paint sample with version 1.0.0 of the ‘paint_sample’ schema, which used a checkbox to add the ‘composition’ metadata. Its metadata may look like this:

```
(‘mgs.paint_sample.colour’, ‘gray’)
(‘mgs.paint_sample.composition’, ‘lime’)
(‘mgs.paint_sample.composition’, ‘iron’)
(‘mgs.paint_sample.__version__’, ‘1.0.0’)
```

After publishing version 2.0.0, in which ‘composition’ is a composite field, the form will be partially filled already, with ‘gray’ as the value of the colour. However, the current values of ‘composition’ will be lost, since there is no such field with that name any more. Instead, we will see a repeatable composite field with two fields inside, for us to fill

[5]This notation is used in order to eventually support semantic versioning [3], but for the moment only major changes are supported.


Paint sample

version 1.0.0 published

version 2.0.0 draft

View Edit Download JSON Discard

Schema ID

paint_sample

Schema label

Paint sample

Add element

Colour of the sample*

other

Add element

Material composition of the pigment

Material

Input type: text

iron

What percentage of the pigment is this material?

As %, between 1 and 100.

Add element

Save draft Publish

sample_001.png

Paint sample

Schema version: 2.0.0

Colour of the sample

other

Material composition of the pigment

Material

iron oxide

What percentage of the pigment is this material?

70

Material

limestone

What percentage of the pigment is this material?

30

Edit Delete metadata for schema "Paint sample"

Figure 2. Create a new version of a schema and, after publishing, modify the metadata.

in with the material and percentage respectively. Because saving the form overwrites all the metadata linked to the schema, the 'composition' fields will be removed and the new fields will be added, resulting in the following metadata:

```
('mgs.paint_sample.colour', 'gray')
('mgs.paint_sample.composition.material', 'lime', '1')
('mgs.paint_sample.composition.percentage', '70', '1')
('mgs.paint_sample.composition.material', 'iron', '2')
('mgs.paint_sample.composition.percentage', '30', '2')
('mgs.paint_sample.__version__', '2.0.0')
```

Figure 2 shows, on the left side, the schema manager, where a new version of the 'paint_sample' schema is being drafted to generate the composite field. On the top left corner, the tab corresponding to the current published version can be seen. Once the draft is published, we can go back to our item and reapply the metadata, which is then rendered as shown on the right side.

In the ManGO metadata schemas manager there are three ways of creating a draft: (1) from scratch, in order to create a whole new schema, (2) derived from a published schema, creating a new version, or (3) derived from a published schema, but creating a different schema. The third case can be used when, for example, we would like to have a schema 'oil_sample' that shares multiple features with 'paint_sample' but has other properties and should follow a different path. In that case, a new schema is created with a different unique name but the same contents and version number "1.0.0"; the specific published version that it comes from is said to be its "parent".

In short, thanks to the life cycle approach, ManGO metadata schemas support both stability and flexibility. On the one hand, stability and documentation are guaranteed by freezing the versions that can be used when applying metadata and registering which version was used at any given moment. On the other, flexibility is allowed by the coexistence and administration of multiple versions, one of which can be a draft, and easily overwriting the incompatible fields when a new published version is used to modify existing metadata. The life cycle status and version number of a schema version are registered in different ways in JSON files, which is the topic of the following section.

JSON FORMAT

Each version of a schema is stored in a JSON file inside a folder with the name of the schema, which is inside a folder for metadata schemas of a given realm. For example, if a project has two schemas, viz. ‘paint_sample’ and ‘canvas’, the contents of the folder in which its schemas are stored would look as follows:

```
schemas
|_paint_sample
|  |_paint_sample-v3.0.0-draft.json
|  |_paint_sample-v2.0.0-published.json
|  |_paint_sample-v1.0.0.json
|_canvas
   |_canvas-v1.0.0-draft.json
```

In this case, there is one (draft) version of the ‘canvas’ schema and three versions of the ‘paint_sample’ schema: 1.0.0 has been archived, 2.0.0 is published and can be used for new metadata, and 3.0.0 is a draft. Thus, the version number is encoded in the name of the file, as well as its status when it is the draft or published version. In addition, the version and status are recorded as attributes of the object contained in the JSON file.

The format of the JSON file is based on the “JSON Schema” format [4] but it deviates from it. The main reason is that the goal of JSON Schema is to describe JSON files, whereas that of metadata schemas is to provide instructions to create forms that will be used to apply metadata[6]. Readers familiar with JSON Schema will be able to identify some similarities, such as the use of `properties` as the attribute collecting the different fields and `type` used to indicate their types, but there are also crucial differences. For instance, from the perspective of forms, `required` is a property of the field that is required or not, whereas JSON Schema lists the required fields as an array that is a property of the schema. Moreover, the `type` attribute in JSON Schema is limited to a range of six primitive types, whereas in metadata schemas it defines the input type in the form (e.g. ‘select’, ‘text’, ‘date’. . .). Metadata schemas also include attributes that are necessary for creating the forms but irrelevant for JSON Schema, such as `multiple` and `repeatable`. With these caveats in mind, let’s look at the specifications of the JSON files used to represent metadata schemas.

The main object in the JSON file representing a version of a schema contains key-value pairs describing the schema itself (the unique name, version and status, a user-facing title, etc.) and a `properties` attribute whose value is another object. The keys of this object are the unique names of the fields, e.g. ‘colour’ or ‘composition’, and the values are objects containing the instructions to render their respective input fields. The example below shows the contents of ‘paint_sample.v-2.0.0-published.json’ with the `properties` attribute collapsed. If, as mentioned before, the schema had been created by copying the contents of another schema instead of from scratch, the `parent` attribute would contain the name and version number of that schema.

```
{
    "schema_name" : "paint_sample",
    "version" : "2.0.0",
```

[6]The Python package mentioned above can validate a JSON-like object against a metadata schema, but it was created after the javascript tool was designed: the main goal of the schemas is still to create forms, not to read JSON objects.

```
    "status" : "published",
    "properties" : {...},
    "title" : "Sample of paint",
    "edited_by" : "username",
    "realm" : "project_collection",
    "parent" : ""
}
```

The contents of the properties attributes would look like this:

```
{
    "colour": {
        "type": "select",
        "multiple": false,
        "ui": "dropdown",
        "values": ["red", "yellow", "blue", "grey", "other"],
        "title": "Colour of the sample",
        "required": true,
        "default": "other"
    },
    "composition": {
        "type": "object",
        "title": "Material composition of the pigment",
        "properties": {
            "material": {
                "type": "text",
                "title": "Material",
                "pattern": "^[a-z ]+$",
                "placeholder": "iron"
            },
            "percentage": {
                "type": "integer",
                "title": "What percentage of the pigment is this material?"
                "minimum": "1",
                "maximum": "100",
                "help": "As %, between 1 and 100."
            }
        },
        "repeatable": true
    }
}
```

Each object representing a field has at least two attributes: `type` and `title`. The `title` is a user-facing title that is shown as a label in the form, whereas the unique name of the field (e.g. ‘colour’, ‘material’, ‘percentage’), used in the actual name of the AVU with namespacing, is only visible as a tooltip. The `type` indicates the type of input field: for composite fields this is “object”, for multiple-choice fields it is “select”, and for simple fields it can be one of “text”, “textarea”, “integer”, “float”, “date”, “time”, “datetime-local”, “email”, “url” or “checkbox”[7].

[7]This checkbox is a single checkbox that has value `true` when checked, for simple boolean values.

Some attributes are optional and can be part of virtually any field:

- `required` indicates whether the field is required. Composite fields cannot be required, but its components can. If `required` is `true`, an additional `default` attribute can specify a default value that is used when the user does not provide an alternative.
- `repeatable` indicates whether the field can be repeated. This is irrelevant in the case of multiple-choice fields, since the same effect is achieved by selecting a multiple-value (e.g. checkbox) instead of single-value (e.g. radio).
- `help` provides a description to be inserted between the label and the input itself in order to explain what the metadata field represents and how it should be filled.

In addition, the different types of fields can have specific attributes that provide additional instructions. In the case of multiple-choice fields, three more attributes are required: `multiple`, `ui` and `values`. When `multiple` is `true`, the user can choose one or more of the options provided; when it is `false`, they can choose at most one. Only the latter type of input can be required and have a default value. The `ui` attribute indicates how the field should be rendered in the form and can have one of three values: 'dropdown', 'checkbox' (when `multiple` is `true`) or 'radio' (when `multiple` is `false`). Finally, `values` is an array of at least two items with the possible values of the metadata field. Therefore, in the example above, 'colour' is rendered as a required dropdown with five options, one of which ("other") is the default value that gets chosen if the user does not select an alternative.

Simple fields can have other options. Those with type "text", "email" or "url" can have a `pattern` attribute with a regular expression used in validation; both that group and numeric inputs can have a `placeholder` attribute with an example value, and numeric inputs only can also have `minimum` and/or `maximum` attributes indicating a range of possible values. In the example, 'material' is then rendered as a text input that must match a pattern requiring the value to only contain lower case letters and spaces, and the example 'iron' is provided (which is not a default value!); 'percentage' is a numeric input field that only accepts integers between 1 and 100. Neither of them are required.

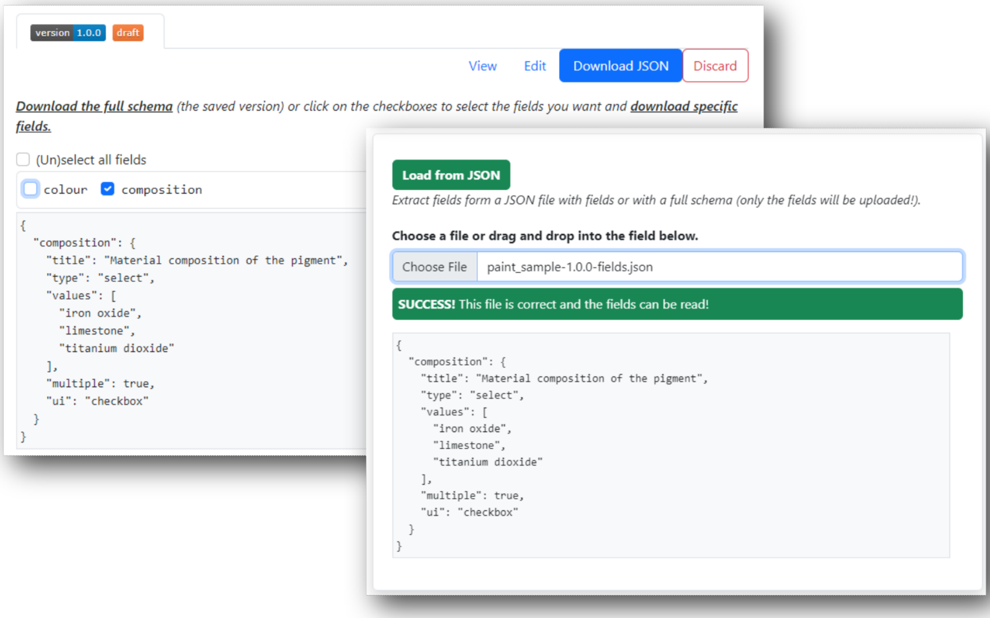
Finally, because composite fields are like miniature schemas, their representation combines features of both fields and schemas. Next to the compulsory `type` (which must be 'object') and `title` and the optional `help` and `repeatable`, they must include a `properties` attribute that follows the same requirements as the `properties` attribute of a schema. In this case, we can see that 'composition' is a repeatable composite field with two components: 'material' and 'percentage'.

The ManGO metadata schema manager exposes this structure and allows users to download JSON files containing a full metadata schema, only the object of the `properties` attribute, or even an object with a subset of the `properties` attribute. The manager also allows users to upload any of those files in order to reuse existing fields in a different schema. For example, say that we don't want the 'material' subfield of composition to be a string, but a checkbox as before, and we have already put a lot of work in creating the list of options for that old checkbox. We can then download the JSON version of the checkbox and then upload it again when designing the new composite field (see Figure 3).

CONCLUSION

In the spirit of making research data FAIR from the start (that is, findable, accessible, interoperable and reusable), we have created a tool to facilitate the application of rich standardized metadata in iRODS. Via namespacing, metadata schemas can bring a coherent set of related metadata together, ensuring unique and consisting attribute names, and further enabling hierarchical structures. Moreover, the metadata schema manager provides a user-friendly interface to create, manage and apply metadata schemas, supporting multiple versions and thus both guaranteeing stability and allowing continuous development of the schemas.

In this paper we have described three main aspects of the metadata schemas: (1) the namespacing as a solution to most of the challenges, (2) the life cycle and (3) the JSON format in which they are stored. One point was briefly



Download the full schema (the saved version) or click on the checkboxes to select the fields you want and *download specific fields.*

☐ (Un)select all fields

☐ colour ☑ composition

```
{
  "composition": {
    "title": "Material composition of the pigment",
    "type": "select",
    "values": [
      "iron oxide",
      "limestone",
      "titanium dioxide"
    ],
    "multiple": true,
    "ui": "checkbox"
  }
}
```

Extract fields form a JSON file with fields or with a full schema (only the fields will be uploaded!).

Choose a file or drag and drop into the field below.

paint_sample-1.0.0-fields.json

SUCCESS! This file is correct and the fields can be read!

```
{
  "composition": {
    "title": "Material composition of the pigment",
    "type": "select",
    "values": [
      "iron oxide",
      "limestone",
      "titanium dioxide"
    ],
    "multiple": true,
    "ui": "checkbox"
  }
}
```

Figure 3. Download a field as JSON and reload it to import it into a new (version of a) schema.

mentioned in the third section and bears repeating, as a likely shortcoming of our current workflow: the JSON format of these schemas in itself is not a standard. It has been tailored to the goal of creating user-friendly forms, which makes conversion between it and other formats less than straightforward. However, we believe that this format is simple enough, and that incompatibilities between it and other formats are more due to the different goals than to formatting choices.

In conclusion, we hope that many researchers using iRODS, either within ManGO or other applications, start using and developing metadata schemas to manage and organize rich, standardized metadata and improve their data description practices.

ACKNOWLEDGMENTS

The work described in this paper was developed within the Research Data Management team of the ICTS department at KU Leuven and received support of the VSC (Flemish Supercomputer Center), funded by the Research Foundation - Flanders (FWO) and the Flemish Government.

REFERENCES

[1] Borgermans, P.: iRODS Python/PRC based portal and tools for active data support in research contexts. In: iRODS User Group Meeting 2023, Leuven, Belgium

[2] iRODS Consortium: "Metadata". iRODS Docs, `https://docs.irods.org/4.2.10/system_overview/glossary/#metadata`, Visited last on 02.06.2023

[3] Preston-Werner, T.: Semantic Versioning 2.0.0, `https://semver.org`, Visited last on 02.06.2023

[4] OpenJS Foundation: JSON-schema, `https://json-schema.org/`, Visited last on 02.06.2023