Providing validated, templated and richer metadata using a bidirectional conversion between JSON and iRODS AVUs

J. Paul van Schayck
MUMC+ DataHub
P. Debyelaan 25,
Maastricht, The
Netherlands
p.vanschayck@maastrichtuniversity.nl

Ton Smeele
Utrecht University
ITS/RDM
Heidelberglaan 8,
Utrecht, The Netherlands
a.p.m.smeele@uu.nl

Daniël Theunissen
MUMC+ DataHub
P. Debyelaan 25,
Maastricht, The
Netherlands
d.theunissen@maastrichtuniversity.nl

Lazlo Westerhof
Utrecht University
ITS/RDM
Heidelberglaan 8,
Utrecht, The Netherlands
l.r.westerhof@uu.nl

ABSTRACT

A frequently recurring question in research data management is to structure metadata according to a standard and to provide the corresponding user interface to it. This has only become more urgent since the introduction of the FAIR principles which state that metadata should use controlled vocabularies and meet community standards.

The iRODS data grid technology is well positioned as a core layer within an infrastructure to manage research data. One of its strengths is the ability to attach any number of attribute, value, unit (AVU) triples as metadata to any iRODS object. This makes iRODS adaptable to very diverse use cases in research data management. However, the challenge of working with more structured metadata is not being addressed by the default capabilities of iRODS. Our aim is to develop a new method for storing richer, templated and validated metadata in AVUs.

JSON is a popular, flexible and easy to use format for serializing (nested) data, while maintaining human and developer readability. Furthermore, a JSON Schema can be used to validate a JSON structure and it can also be used to obtain a dynamically generated form on the basis of this schema. This combination of functionalities makes it an excellent format for metadata. We have therefore designed and implemented a bidirectional conversion between JSON and AVUs. The conversion method has been implemented as Python iRODS rules that allow to set and retrieve AVU metadata on an iRODS object using a JSON structure. Optionally, a policy can be installed to validate metadata entry and updates against the JSON Schema that governs the object.

With this work we provide other iRODS developers with a generic method for conversion between JSON and AVUs. We are encouraging others to use the conversion method in their deployments.

Keywords

Metadata, AVUs, JSON, conversion, validation, presentation

INTRODUCTION

Over recent years there has been an increasingly urgent call for the practice of open science [1]. Driven by the core values in research of transparency and reproducibility, the sharing of research data has become a hot topic.

Additional reasons for sharing research data are to better leverage investments in research by promoting reuse and making those data that can be considered public assets, available to the public [2]. To facilitate this accountability and reuse of research data, the Findable, Accessible, Interoperable and Reusable (FAIR) principles of research data have been introduced and broadly taken up [3]. Briefly, the FAIR principles suggest for research data to be globally and uniquely identifiable, and associated with searchable metadata (”Findable”); these identifiers should point to (meta)data using an open protocol (”Accessible”) and that this data uses a formal representation language using widely applicable ontologies (”Interoperable”); finally, data should be provided with cross-references, provenance and license information (”Reusable”). Furthermore, the FAIR principles state that all this should be provided in both human and machine-readable form to facilitate automated pipelines and the increasing need for automated analysis and large-scale data research. However, the FAIR guidelines did not define any form of implementation recommendations for these principles.

The iRODS data grid technology is well-positioned as a core layer within an infrastructure to manage research data [4]. iRODS features support for preservation properties that are important for research data management such as authenticity, integrity and chain of custody. Most of these properties are met using metadata to annotate data objects and collections. Within iRODS, a basic building block for managing metadata is the AVU, short for *Attribute-Value-Unit*. The name ”AVU” refers to its compound structure: It consists of three string-typed fields that as a triple represent a single metadata property of the data. While typically more than one AVU is needed to communicate and to document properties of research data[1], currently the iRODS support for AVU *composition* is somewhat limited. Microservices operate either on a single AVU or on an attribute-value map structure that does not allow a unit component to be specified. Attributes with multiple values, nested structures and atomic operations on a composition of AVUs are not supported. There is also no way to specify a template or structure that AVUs associated to an object should adhere to. These limitations may hinder the implementation of the FAIR principles or could lead to *ad hoc* solutions that are not transferable to other systems.

In contrast, many applications have adopted the data interchange standard JavaScript Object Notation (JSON) to efficiently exchange and operate on composed data structures [5]. JSON is lightweight and can be used across many programming languages. JSON data structures consist of an unordered collection of name/value pairs referred to as an *object*. Values are primitive data types, or an ordered list of values, or another JSON object. To improve interoperability, in 2017 IETF has published a more restrictive version of the JSON standard [6]. For instance, this version requires the name of name/value pairs to be unique, so that programming languages can conveniently implement JSON using map constructs.

JSON can be used to serialize arbitrary metadata, resulting in an equally arbitrary set of AVUs. For the purpose of adhering to the FAIR principles however, we seek to restrict the metadata that documents research data to a well-defined set of composed, related and consistent properties. The semantics of this metadata structure can be modeled as a *template*. Such a template can be applied to validate operations that attempt to modify any of the AVUs within the *namespace* of the template.

We propose to use JSON Schema as a technique to represent a metadata template within the context of iRODS. The metadata template will act on a composition of AVUs that is represented by a JSON data structure. JSON Schema is a draft internet standard that aims to define the structure and content of JSON formatted data [7]. Using a JSON Schema definition, applications such as iRODS can validate and interact with instances of JSON formatted data.

The application of a JSON Schema based template does not need to be limited to iRODS server-side validation. Figure 1 shows how client applications can opt to use the same JSON Schema in a presentation layer to *dynamically* render forms that facilitate data entry of metadata. For instance the React[2] component react-jsonschema-form[3] is

[1]see for instance the DataCite specification at https://schema.datacite.org/
[2]see https://reactjs.org
[3]see https://github.com/rjsf-team/react-jsonschema-form

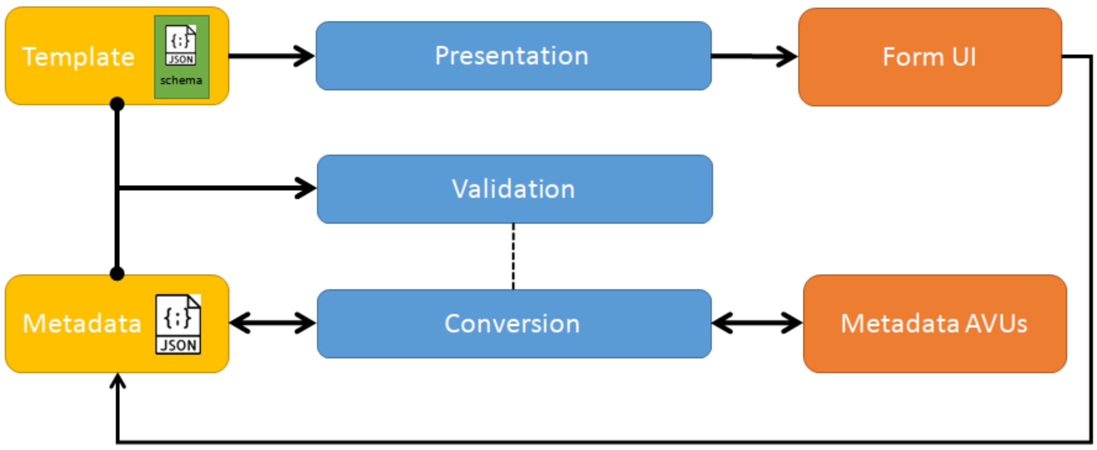

Figure 1. Overview of the schematic layers between JSON, AVUs, JSON-schema and its process in conversion, validation and presentation.

used by Utrecht University, The Netherlands to render metadata entry forms in its data management application Yoda [8]. DataHub at MUMC+ and Maastricht University, The Netherlands seeks to implement a similar solution based on the CEDAR Workbench [9]. DataHub's use case is focused on semantically linked (meta)data. Therefore not only should the JSON structure be governed by a JSON Schema, in addition its vocabulary and structure must conform to the W3C JSON-LD recommendation [10].

Hence our research can be applied on three levels. At the foundation level, a conversion method supports the use of JSON to manage arbitrary compositions of AVUs in iRODS and to exchange these compositions efficiently with client applications (Methods section). The optional second level validates the exchanged JSON against a metadata template defined in JSON Schema. Both the first and second levels are implemented in the iRODS server (Results section). A third level can optionally be implemented as part of a client application. It uses the (same) metadata template for dynamic form-based user interactions. In the Discussion section, the main advantages and disadvantages we found using the proposed methods are discussed. Finally, the research results are summarized and an outlook is provided in the Conclusion section.

METHODS

Bidirectional conversion between JSON and AVU structures

We intend to represent a set of iRODS AVUs as a JSON structure and vice versa and use the serialized data in communications between the iRODS server and client applications.

Design goals

Before creating the conversion method we set five design goals.

1. The conversion method must be a bijective function to ensure lossless conversions between JSON and AVU structures in both directions. Any JSON structure that is compliant with the JSON specification should be supported. The method should provide support for Unicode characters, nested structures, and ordered lists.
2. It must be easy to identify corresponding JSON objects and AVU attributes. This means that, especially for simple JSON structures, it should be trivial to retrieve a JSON element from the AVUs without first back-converting the JSON.

3. The conversion method should be lean and efficient. We seek to avoid an explosion in the number of AVUs as a result of representing a nested JSON structure.
4. The method should be compatible with existing use cases that operate directly on AVUs.
5. The conversion method should be compatible with JSON-LD use cases.

Conversion method specifications

Using the design goals set as requirements we arrived, over several iterations, at a working design for the conversion. An example JSON structure and its converted counterpart in AVUs is listed in Table 1. This example will be used to explain how the conversion method works. Further examples can be found in the online repository[4].

Table 1. Example of a JSON structure and its converted counterpart in AVUs.

```
{
  "title": "Hello World!",
  "parameters": {
    "size" : 42,
    "readOnly" : false
  },
  "authors" : ["Foo", "Bar"],
  "references": [
    {
      "title": "The Rule Engine",
      "doi": "1234.5678"
    }
  ]
}
```

Representation in JSON

Attribute	Value	Unit
title	Hello World!	root_0_s
parameters	o1	root_0_o1
size	42	root_1_n
readOnly	False	root_1_b
authors	Foo	root_0_s#0
authors	Bar	root_0_s#1
references	o2	root_0_o2#0
title	The Rule Engine	root_2_s
doi	1234.5678	root_2_s

Representation in AVUs

The proposed conversion method repurposes the unit field of the AVU to encode JSON variable type and structure information. This also reduces the chance of collisions with existing AVUs that presumably have an empty unit field. Currently, nearly all the iRODS microservices that facilitate AVU operations, for example `msiAssociateKeyValuePairsToObj`, do not allow rule developers to specify content for the unit field. As a result of this limitation, the unit component of the AVU is hardly ever used at the time of writing.

The AVU unit field comprises of four components, separated by an underscore character, except for the last component where a hash is used. The first component indicates a *namespace* carried by all the AVUs that belong to this set. Conversion operations will only affect AVUs that are part of the selected namespace. In addition, it facilitates that iRODS objects are annotated with multiple JSON structures, each identified by their own namespace. In example Table 1 the namespace is `root`.

The second component is an *object sequence number* that keeps track of AVUs that are part of the same JSON object. The top level JSON object is assigned sequence number ”0”. In the example this includes `title`, `parameters`, `authors` and `references`. Note that the element `parameters` holds a nested object as its value. The next sequence number ”1” is assigned to this nested object and note the sequence number in its (otherwise unused) AVU value component, prefixed by the character `o`. All elements of the nested object, in this example `size` and `readonly`, have the object sequence number ”1” in their unit field.

[4]see https://github.com/MaastrichtUniversity/irods_avu_json

An important design goal of the conversion method is the support of different variable types within JSON. AVUs only allow string values, while JSON supports various primitive types. The third component of the AVU unit field is used to indicate the JSON *type* of the value. See table 2 for an overview of the supported types. A special case is the `empty array` type that indicates the presence of an array without any members. To achieve a lean conversion, we only create AVUs to represent *members* of an array. We have to make an exception for an empty array, which otherwise would not have any AVU representation at all. Without this provision, a later conversion from AVU back to JSON would not be able to recreate the empty array.

Type	AVU unit-type	AVU value	Remarks
string	s	The literal string	
object	o + object_id	o + object_id	The AVU value field is not used for conversion
boolean	b	"True" or "False"	
number	n	String value of float or int	
null	z	"."	AVUs do not allow empty values
empty string	e	"."	AVUs do not allow empty values
empty array	a	"."	For convenience during conversion. See text.

Table 2. Overview of JSON variable types and their corresponding type string.

The fourth, optional component of the AVU unit field is used to denote an *ordered index* of the element. This component is separated from its predecessor using a hash character `#`. The JSON specification includes the array type. This is an ordered list of elements of any type. As AVUs are unordered, the last component of the unit field denotes the array index to maintain order in arrays.

Summarizing, the AVU unit field has been used for the following purposes: 1. defining the JSON namespace, 2. the object sequence number, 3. the value type and 4. the array index. A regular expression for capturing these components of the unit field is shown in Listing 1.

```
^([a-zA-Z0-9_])+_([0-9]+)_([osbanze])((?<=o)[0-9]+)?((?:#[0-9]+?)*)
```

Listing 1. A regular expression to parse the components of the unit field

Validation of JSON structure using a JSON Schema template

The conversion method discussed above allows client applications to store JSON structures efficiently within the iRODS server. This method is agnostic to the structure and the semantics of the metadata that is exchanged. For some use cases this may suffice. Many use cases however require that metadata stored or exchanged is compliant with a certain standard. We will now propose a validation method to fulfill this need.

Design goals

The validation method needs to meet three design goals.

1. It must be able to assess that a *stored or exchanged set* of metadata meets predefined quality levels with respect to structure and semantics as typically documented in metadata standard. The metadata schema can vary per iRODS object. For instance, objects that belong to a data set related to the Geosciences may require geospatial location annotations whereas objects related to History disciplines may depend on chronological classification metadata.

2. It should also be possible to annotate a single object with multiple disjunct sets of metadata. The metadata template can vary per set of metadata.

3. In line with the conversion method, the validation method should again be compatible with existing use cases that operate directly on individual AVUs.

Validation method specifications

For the purpose of validation, we shall consider sets of metadata that annotate an iRODS object rather than individual AVUs. These sets are easily identified by their namespace identifier which is incorporated in the unit component of the AVU. This makes the validation compatible with existing use cases as AVUs without a namespace will not be affected by the validation and protection policies.

We select the draft internet standard JSON Schema to specify a metadata template used to validate sets of metadata [7]. JSON Schema conveniently supports the description of quality properties of individual metadata elements as well as qualities that span across elements, for instance dependency relationships. Incoming metadata will be in JSON representation and can be validated directly against a template. An example of the resulting schema is shown in Listing 2.

```
{ "$id": "http://example.com/myschema.json",
  "$schema": "http://json-schema.org/schema#",
  "type": "object",
  "additionalProperties": false,
  "properties": {
    "title": { "type": "string" },
    "parameters": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "size": { "type": "number" },
        "readOnly": { "type": "boolean" }
    }},
    "authors": {
      "type": "array",
      "items": { "type": "string" }
    },
    "references": {
      "type": "array",
      "items": {
        "type": "object",
        "additionalProperties": false,
        "properties": {
          "title": { "type": "string" },
          "doi": { "type": "string" }
}}}}}
```

Listing 2. JSON schema that corresponds to the JSON structure of Table 1.

RESULTS

Conversion method implementation

The implementation has been developed for iRODS version 4.2.x [11]. Since iRODS 4.2 does not expose any microservices to modify the unit field of an AVU triple, custom iRODS microservices have been developed. The conversion scheme has been developed as Python 2.7 and Python 3 module[5] named `irods_avu_json`. The outcome of the conversion of the example JSON is shown in Table 1.

The `irods_avu_json` module has in itself no interaction with or dependency on iRODS. To expose this functionality within an iRODS installation we developed an iRODS ruleset [6]. The developed ruleset uses the Python Rule Engine recently released with iRODS 4.2 to import the irods_avu_json module.

An overview of the functionality being exposed by the ruleset is summarized in Table 3. All ruleset functions allow specifying any iRODS object type (collection, object, user, group or resource) in a similar fashion as that iRODS AVUs can be attached to any iRODS object. Furthermore, all functions also expect the JSON namespace to know which AVU set to operate on.

Validation method implementation

The special *$schema* AVU denotes whenever a JSON Schema is attached to an iRODS object. We implemented two ways for the JSON Schema to be specified in the value field of this AVU. (1) An (public or private) URI pointing to the stored JSON schema. Optional caching of this URI has been implemented for performance reasons. (2) By specifying 'i:' in front of a path the JSON Schema is directly retrieved from within iRODS. Care must be taken that this iRODS object is accessible for anyone allowed to modify the iRODS object to which the JSON Schema is attached. Other methods for storing the JSON Schema could be devised and implemented at a later point.

Note that both the iRODS server and client applications can benefit from using the metadata template to check the validity of any metadata that is being exchanged. Therefore a best practice is that the template reference is an absolute URI and the metadata template itself is available at an internet-accessible location.

Validation of the JSON structure set by `setJsonToObj()` is triggered by the presence of the `$schema` attribute and the same JSON namespace being present on the iRODS object. After retrieving the JSON Schema contents, the validation is performed using the `jsonschema` Python module. Any validation errors will be passed back to the caller of `setJsonToObj()`. To ensure only the full and validated JSON object is being stored first all existing AVUs of the same JSON namespace are removed before the new one are set. This also ensures that any no longer existing parts of the JSON object are removed during a `setJsonToObj()` operation.

The irods_avu_json-ruleset further implements validation of a JSON object by implementing a policy enforcement points (PEPs), which are executed during the modification of AVUs. Whenever a *$schema* AVU is present on the iRODS object and the AVU being modified is part of the same JSON namespace the operation is disallowed. Modification of these AVUs can only be performed through `setJsonToObj()`. Because `setJsonToObj()` would also trigger the same PEPs, the PEPs check whether execution is coming from `setJsonToObj()` and has been validated against the JSON Schema. This is achieved through setting a Python global variable that is preserved in the rule engine memory.

Function	Description
setJsonToObj(object, objectType, jsonNs, json)	Set a JSON to an iRODS object.
getJsonFromObj(object, objectType, jsonNs)	Retrieve a JSON from an iRODS object.
getJsonSchemaFromObj(object, objectType, jsonSchema, jsonNs)	Attach a JSON Schema to an iRODS objec.
setJsonSchemaToObj(object, objectType, jsonNs)	Retrieve a JSON Schema from an iRODS object.

Table 3. Functionality exposed by the irods_avu_json-ruleset

[5]see https://github.com/MaastrichtUniversity/irods_avu_json
[6]see https://github.com/MaastrichtUniversity/irods_avu_json-ruleset

DISCUSSION

We successfully used the conversion method and the accompanying validation method in several pilot use cases. We found several limitations and problems with the current implementation of the conversion and validation method.

The implementation of the conversion method currently requires all existing AVUs relating to a JSON namespace to be removed before the new JSON is added. This may lead to performance issues with very large JSON structures. Furthermore, the addition of all JSON related AVUs is not an atomic operation, meaning that collisions may occur if multiple clients modify the same iRODS object at once.

The current implementation of validation uses the metadata PEPs to prevent any non-validated AVUs to be created or modified. These PEPs directly wrap around their AVU modification microservice counterparts. Therefore a single microservice call can, using a wildcard, operate on multiple AVUs. This means logic created for the PEPs is rather convoluted. Furthermore, the chosen implementation of using a global Python variable to bypass the PEPs when `setJsonToObj()` is being called breaks when the call for `setJsonToObj()` is initiated from a catalog consumer instead of the catalog provider. This is because in that case of the call being initiated from the catalog consumer the global variable is not in memory when the PEP is being executed on the catalog provider.

The performance issue, the non-atomic operation of the current conversion and the issue with the PEPs can all be tackled by the introduction of a multi-AVU atomic core iRODS functionality. Such a microservice could handle the entire operation of converting a JSON structure and its validation at once.

While not directly shown in this work the use of JSON Schema can be extended beyond the validation level that is currently implemented. As mentioned before, an important feature is the auto-generation of web forms from the JSON Schema. Several implementations of libraries capable of this functionality exists and we have explored several of those. Furthermore, we envision that the use of JSON Schema for presentation can be further extended to for example auto-generated search forms.

CONCLUSION

We set out to add a generic toolset to iRODS for handling richer, templated and validated metadata. We have developed a bidirectional conversion between JSON and AVUs and validation provided through JSON Schema. Both methods have been validated to meet their design goals via a proof of concept followed by an application-level implementation. The methods developed provide new starting points for iRODS developers in facilitating research data management that adheres to the FAIR principles.

ACKNOWLEDGMENTS

This work was sparked by the discussions in the iRODS metadata template working group and we would like to thank them for their feedback. We would like to thank all members of the DataHub Maastricht team for their helpful feedback and comments. Finally we thank Raimond Ravelli, Peter Peters and Michel Dumontier for their critical reading of this manuscript.

REFERENCES

[1] M. R. Munafò, B. A. Nosek, D. V. Bishop, K. S. Button, C. D. Chambers, N. P. Du Sert, U. Simonsohn, E.-J. Wagenmakers, J. J. Ware, and J. P. Ioannidis, "A manifesto for reproducible science," *Nature human behaviour*, vol. 1, no. 1, p. 0021, 2017.

[2] C. L. Borgman, *Big Data, Little Data, No Data, Christine L. Borgman, The MIT Press, Cambridge, MA (2015), Xxiv, 383 p. $70.00, ISBN: 978-0-262-02856-1.* Elsevier, 2015.

[3] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, and others, "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific data*, vol. 3, 2016.

[4] R. Moore, “Towards a theory of digital preservation,” *International Journal of Digital Curation*, vol. 3, no. 1, 2008.

[5] ECMA, “ECMA-404: The JSON Data Interchange Syntax,” *Ecma International*, vol. Standard, no. Second edition, 2017.

[6] T. Bray, “IETF RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format,” *Internet Engineering Task Force (IETF)*, 2017.

[7] H. Andrews and A. Wright, “JSON Schema A Media Type for Describing JSON Documents,” *Internet Engineering Task Force (IETF)*, vol. Internet-Draft, Mar. 2018.

[8] T. Smeele and L. Westerhof, “Using iRODS to manage, share and publish research data: Yoda,” in *Proceedings of the 2018 iRODS User Group Meeting*, (Durham NC), University of North Carolina, June 2018.

[9] R. S. Gonçalves, M. J. O’Connor, M. Martínez-Romero, A. L. Egyedi, D. Willrett, J. Graybeal, and M. A. Musen, “The CEDAR Workbench: an ontology-assisted environment for authoring metadata that describe scientific experiments,” in *International Semantic Web Conference*, pp. 103–110, Springer, 2017.

[10] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, “JSON-LD 1.0,” *W3C Recommendation*, vol. 16, p. 41, 2014.

[11] A. Rajasekar, R. Moore, M. Wan, and W. Schroeder, “Policy-based Distributed Data Management Systems,” *Journal of Digital Information*, vol. 11, no. 1, 2010.