iRODS®
UGM 2023
Chapel Hill

USER GROUP MEETING
2023 PROCEEDINGS

PUBLISHED BY THE iRODS CONSORTIUM

iRODS
User Group Meeting 2023
Proceedings

15TH ANNUAL CONFERENCE SUMMARY

The iRODS User Group Meeting of 2023 gathered together iRODS users, Consortium members, and staff to discuss iRODS-enabled applications and discoveries, technologies developed around iRODS, and future development and sustainability of iRODS and the iRODS Consortium.

The in-person and virtual four-day event was held from June 13th to 16th, hosted by the iRODS Consortium in Chapel Hill, NC, with 91 people attending from 11 countries. Attendees and presenters represented over 37 academic, governmental, and commercial institutions.

TALKS AND PAPERS

LIGHTNING TALKS

What to tell about RDM to whom?

Ander Astudillo – SURF

Yoda and Rspace deployment and the Liebniz Institute on Aging

Rory Macneil – Research Space

Lazlo Westerhof – Utrecht University

Teaching old dog new tricks: Fun with iRODS at CyVerse 2023 edition

Nirav Merchant – CyVerse / University of Arizona

Beyond Data Management with Globus

Vas Vasiliadis – Globus

Azure as native storage plugin (Like S3)

Jan Graaf – Netherlands Cancer Institute (NKI)

What would we like from a Rust iRODS client library?

Phillip Davis – iRODS Consortium

Let’s bring light to dark data with iRODS: Come write a new proposal with CyVerse

Nirav Merchant – CyVerse / University of Arizona

Dataverse integration dashboard: pulling data from iRODS

Ingrid Barcena Roig – KU Leuven

iBridges: A comprehensive way of interfacing with iRODS

Christine Staiger
Utrecht University
c.staiger@uu.nl

Tim van Daalen
Wageningen University
tim.vandaalen@wur.nl

John Mc Farland
University of Groningen
mcfarland@astro.rug.nl

ABSTRACT

iRODS is a rich middleware providing means to facilitate data management for research. It implements all necessary concepts like resources, metadata, permissions, and rules. However, in research most of the concepts are still new. Hence, researchers and their support staff are challenged using the current interfaces and tools to 1) learn about those concepts and 2) familiarise themselves with the different APIs and command line interfaces. This creates the need for a steep learning curve for researchers and research supporters, slowing down the adoption of iRODS. To ease the usage of iRODS we present iBridges.

iBridges is a standalone desktop application, written in Python, to provide users of Windows, Linux, and MacOS with a graphical user interface (GUI) to interact with iRODS servers. The tool is agnostic to any rules/policies in the server. Out-of-the-box iBridges supports three main functions: browsing and manipulating data objects, upload/download data, and searching data collections.

Research data management still is an evolving topic for which new tools are constantly developed. To allow for easy creation of workflows employing other services and to integrate them with data managed in iRODS, we kept the code as modular and simple as possible. This also allows to add new features as the understanding and development in research data management progresses.

We demonstrate how such an integration works for an electronic lab notebook and an audio transcription tool.

10 years at CyVerse: Some iRODS Administration Practices

Tony Edgin
CyVerse / University of Arizona
tedgin@cyverse.org

ABSTRACT

iRODS is extremely flexible in its configuration. Furthermore, it is not an opinionated system. This makes it powerful, but it also makes it difficult to manage. During this talk I will present a few practices and lessons I've learned over the last ten years maintaining CyVerse's iRODS grid. These practices will include how to decommission a storage server with no or minimal downtime, how to asynchronously replicate data to an off-site resource server, and how to transfer large sets of small files more quickly. Afterwards, I will offer to organize an interest group that would meet periodically to discuss best practices for iRODS administration.

iRODS UGM 2023, June 13-16, 2023, Chapel Hill, NC

ManGO: A web portal and framework built on top of iRODS for active research data management

Paul Borgermans
KU Leuven
paul.borgermans@kuleuven.be

Mariana Montes
KU Leuven
mariana.montes@kuleuven.be

Ingrid Barcena Roig
KU Leuven
ingrid.barcenaroig@kuleuven.be

ABSTRACT

At the University of Leuven. Belgium, we are building the infrastructure and software layers to leverage iRODS as a major building block in active research data management. This involves various workflows and processing of data and metadata during the lifetime of a research project. One of the important components consists of a modular and adaptable web portal built using the iRODS Python client. Given the wide range of use cases, the web framework employs some classical architectural patterns to decouple specialised domain specific needs from the core system. It also has features that make it behave like a content management system, including a (view) template override system that make the representation of collections and data objects dependent on for example specific metadata or collection structure. Metadata is a prime focus to steer many aspects of this framework along its core use for research data, and a considerable effort was also put in a user friendly metadata schema management system. In this talk, we will present the current status as well as near future plans.

iRODS Object Store on Galaxy Server: Application of iRODS to a Real Time, Multi-user System

Kaivan Kamali, Nate Coraor, John Chilton, Marius van den Beek, Anton Nekrutenko
Penn State University
kxk302@gmail.com

ABSTRACT

Galaxy (https://galaxyproject.org) is an open-source platform for data analysis that enables users to 1) Use tools from various domains through its graphical web interface, 2) Run code in interactive environments such as Jupyter or RStudio, 3) Manage data by sharing and publishing results, workflows, and visualizations, and 4) Ensure reproducibility by capturing the necessary information to repeat data analyses.

To store data Galaxy utilizes ObjectStore as its data virtualization layer. It abstracts Galaxy's domain logic for data persistence technology. Currently, Galaxy mainly uses a Disk ObjectStore for data persistence. To extend Galaxy's data persistence capabilities, we had previously extended Galaxy's ObjectStore to support iRODS. In this work, we discuss the steps in deploying iRODS Object Store on the USA-based Galaxy server (usegalaxy.org) and the challenges we faced. To the best of our knowledge, after CyVerse (https://cyverse.org/about), this is one of the few application of iRODS to a real time, multi-user system.

iRODS UGM 2023, June 13-16, 2023, Chapel Hill, NC

iRODS HTTP API

Kory Draughn
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
korydraughn@renci.org

Terrell Russell
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
unc@terrellrussell.com

ABSTRACT

The iRODS Protocol has remained relatively static for more than 20 years. This is a testament to its original planning, but also means any redesign would carry a heavy upgrade and migration cost. Additionally, the protocol is novel to most developers and differs in implementation across client programming languages which hurts both approachability and adoption. This paper covers the design, implementation, and early performance results of a new HTTP API for interacting with iRODS.

Keywords

iRODS, http, api, cpp

WHAT AND WHY

The new iRODS HTTP API [1] is an experimental redesign of the relatively recent iRODS C++ REST API [2] originally developed and released in 2021. The HTTP API has renewed focus on consistency and familiarity, as well as maintainability into the future. The main goals of the project are to:

- Present a cohesive representation of the entire iRODS API over the HTTP protocol, effectively simplifying development of client-side iRODS applications for new developers. If all iRODS functionality is exposed via HTTP, then a significant number of existing developers can work with iRODS without having to climb a relatively steep learning curve.
- Maintain performance close to the iCommands. The iCommands have been around for nearly 20 years and have proven efficient for their focused responsibilities. Any new solution should have a similar (if not better) performance profile.
- Remove behavioral differences between client-side iRODS libraries by building new libraries on top of the HTTP API (including C, C++, Java, and Python). Since most languages now provide standard libraries around making HTTP calls, language-specific wrappers that call the HTTP API could be very thin, proving both easy to produce and easy to maintain.
- Potentially be absorbed by the iRODS server if adoption is significant. The world has chosen HTTP as the most-favored protocol for getting things done, and apart from large data movement, it should be considered as the primary candidate protocol for exposing the functionality of an iRODS namespace.

The iRODS C++ REST API proves that presenting iRODS as HTTP is possible, however, usage of the project over the last couple of years has uncovered some challenges.

iRODS UGM 2023 June 13-16, 2023, Chapel Hill, NC

- Too many open ports raise security concerns. The security personnel tasked with asking lots of questions and understanding the traffic on their networks are suspicious of a novel protocol and do not like opening the standard 200+ ports for iRODS parallel transfer.
- Stability issues (e.g. crashing endpoints). This would have gotten better over time, but the relatively high number of moving parts was already feeling brittle.
- Separation of endpoints increases complexity due to multiple layers. The design is good, but appears to be overkill for the work needing to be done today. We may yet end up with more abstraction in the future, but it felt to be too much without a corresponding requirement.
- Pistache HTTP library [3] lacks completeness/maturity/adoption. This iRODS HTTP API project needs to provide a solid foundation that other tools and layers can depend on. Pistache was not going to provide this stability and confidence in the near-term.
- The names of existing endpoints are fairly general which leads to difficulty in naming of new endpoints. Getting an API correct at the beginning is hard, and refactoring is even harder. The current endpoints had already boxed in our ability for providing consistency moving forward.

The iRODS HTTP API is aimed at resolving these issues by taking a different approach based on what we have learned from the community and the iRODS S3 API [4].

DESIGN

With the design goals firmly in place, an initial brainstorming document (in the form of a GitHub gist [5] in April 2023) laid out a fairly comprehensive listing of this new HTTP API effort. Some of the early decisions included:

- This new API would be run as a single binary exposing one (or two) ports. This will be easier to handle by administrators and be more expected by any interested security teams.
- The library of choice would be Boost.Beast [6], a proven C++ header-only library providing networking facilities for building high performance libraries and applications which need support for HTTP/1 and Websockets. It beat out other candidate libraries and was selected earlier in the year as the basis for the iRODS S3 API and proved the 'right answer'.
- The set of URL endpoints would be fixed and expected for iRODS users. They will represent the 'nouns' of iRODS and be easy to remember and 'guess'.
- The API itself would be renamed from REST to HTTP to reflect the reality that the rules of REST are not clear or consistent and that the statefulness of the iRODS API does not map as well to the singular nature of REST verbs.

API Endpoints and Operations

The endpoints of the new iRODS HTTP API will reflect the well-known nouns of iRODS and provide a stable baseline for exploration and experimentation (as well as documentation). These endpoints are named based on concepts and entities defined in iRODS (in alphabetical order):

- /authenticate
- /collections
- /data-objects
- /info

- /metadata
- /query
- /resources
- /rules
- /tickets
- /users-groups
- /zones

Operations associated with these endpoints are specified via parameters. This decision keeps URLs simple (i.e. top-level only, no nesting required) and allows both new and existing developers to make educated guesses as to which URL exposes the behavior they are interested in.

For example, if a developer wants to modify a user, they would explore `/users-groups`. Or, writing data to a data object would happen through the `/data-objects` endpoint.

It is early days for this new API, but these endpoints should prove to be stable.

API Parameters

At this time, all endpoints, except `/authenticate`, accept an operation (`op`) parameter which is mapped to a function responsible for executing the requested operation. There can be multiple operations on each endpoint and represent the 'verbs' associated with that 'noun'.

The operations available across multiple endpoints will share common values as much as possible (e.g. stat, list, create, remove, etc.). This is designed to make the entire API more predictable, familiar, and approachable for both new and returning developers.

Some of the common parameters used so far include:

- lpath
- replica-number
- src-resource
- dst-resource
- offset
- count

CONFIGURATION

Tools to generate, read, parse, and manipulate JSON files have proven to be available nearly everywhere and has become the standard in the iRODS ecosystem. For these reasons, the iRODS HTTP API will be driven by a single JSON configuration file. It has two halves, one for each of the two 'roles' the API is playing: an HTTP server, and an iRODS client. This format is modeled after NFSRODS [7].

```
{
    // Defines HTTP options that affect how the
    // client-facing component of the server behaves.
    "http_server": {
        // ...
    },

    // Defines iRODS connection information.
    "irods_client": {
        // ...
    }
}
```

http_server

The HTTP server configuration currently allows for controlling the listening host and port, the log level, authentication options, and threading and timeout options. These are the preliminary values and we expect these to change as we explore deployment use cases, optimize for memory and bandwidth, and learn more about how this API plays with other software (including high-availability configurations and reverse proxies for TLS/SSL termination).

```
"http_server": {
    "host": "0.0.0.0",
    "port": 9000,

    "log_level": "warn",

    "authentication": {
        "basic": {
            "timeout_in_seconds": 3600
        }
    },

    "requests": {
        "threads": 3,
        "max_rbuffer_size_in_bytes": 8388608,
        "timeout_in_seconds": 30
    },

    "background_io": {
        "threads": 3
    }
}
```

irods_client

The iRODS client side of the configuration is very straightforward and will change less dramatically as we continue development. It covers the host, port, and zone, as well as the proxy `rodsadmin` information. This API must be deployed as an administrator since it holds a long-running connection pool with the iRODS server and uses the `rc_switch_user` functionality to proxy for regular user operation requests (read more in the next section). The remaining settings are to protect the iRODS HTTP API memory footprint.

```
"irods_client": {
    "host": "<string>",
    "port": 1247,
    "zone": "<zone>",

    "proxy_rodsadmin": {
        "username": "<string>",
        "password": "<string>"
    },

    "connection_pool": {
        "size": 6,
        "refresh_timeout_in_seconds": 600
    },

    "max_rbuffer_size_in_bytes": 8192,
    "max_wbuffer_size_in_bytes": 8192,

    "max_number_of_rows_per_catalog_query": 15
}
```

CONNECTION POOLING

Historically, iRODS clients have assumed a simple back and forth flow of messages with the iRODS server. Each operation traditionally required a connect and disconnect as well as authentication. For many use cases, this can kill performance on anything other than large transfers. This insight led to two enhancements being added to the iRODS 4.3.1 server. First, proxy user support for `irods::connection_pool` and `irods::client_connection`. This is not yet merged, but available as pull request #7047 [8]. Second, a new client call named `rc_switch_user` which allows the identity associated with an `RcComm` to be changed in real-time. This work can be found in pull request #6691 [9].

With these new facilities, the iRODS HTTP API can reuse existing iRODS connections to significantly boost performance.

The following code example shows how a client can use the connection pool, switch the associated user, and return a live connection to the caller.

```
// TODO May require the zone name be passed as well for federation?
auto get_connection(const std::string& _username)
    -> irods::connection_pool::connection_proxy
{
    namespace log = irods::http::log;

    auto& cp = irods::http::globals::conn_pool;
    auto conn = cp->get_connection();
    const auto& zone = irods::http::globals::config->at("irods_client")
        .at("zone").get_ref<const std::string&>();

    log::trace("{}: Changing identity associated with connection to [{}].",
              __func__, _username);

    auto* conn_ptr = static_cast<RcComm*>(conn);
    const auto ec = rc_switch_user(conn_ptr, _username.c_str(), zone.c_str());
    if (ec != 0) {
        log::error("{}: rc_switch_user error: {}", __func__, ec);
        THROW(SYS_INTERNAL_ERR, "rc_switch_user error.");
    }

    log::trace("{}: Successfully changed identity associated with connection to [{}].",
               __func__, _username);

    return conn;
} // get_connection
```

PARALLEL WRITES

Another aspect of providing a middle-tier API is the delicate business of handling parallel connections from the client (and through to the associated iRODS server). iRODS does not allow a data object to be written directly in parallel without coordination (in the form of a replica access token).

iRODS clients wanting to upload data in parallel are required to do the following:

1. Open a stream to the replica of interest.
2. Capture the Replica Access Token from the stream.
3. Open secondary streams.
   - Each stream must use its own connection
   - Each stream must target the same replica
   - Each stream must use the same open flags
   - Each stream must pass the Replica Access Token obtained from the stream in step (2)
4. Send bytes across streams.
5. Close secondary streams without updating the catalog.
6. Close the original stream normally.

This set of steps is now currently fully supported in the iRODS HTTP API through the use of a ”Parallel Write Handle”. However, this means the iRODS HTTP API server maintains state on behalf of the client.

Therefore, performing a parallel write requires the use of two operations. The first, `parallel_write_init`, instructs the server to allocate memory for managing the state of the upload. The second, `parallel_write_shutdown`, instructs the server to deallocate memory obtained via the earlier call to `parallel_write_init`.

Transfers of large files must use `multipart/form-data` as the HTTP content type. Failing to honor this series of steps will result in an error or corrupt data in the iRODS server.

The following example shows how to obtain, use, and then close the parallel write handle:

```
http_api_url="${base_url}/data-objects"

# Open 3 streams to the data object, file.bin.
```

```
transfer_handle=$(curl -H "Authorization: Bearer $bearer_token" "$http_api_url" \
  --data-urlencode 'op=parallel_write_init'                                      \
  --data-urlencode "lpath=/tempZone/home/rods/file.bin"                          \
  --data-urlencode 'stream-count=3'                                              \
  | jq -r .parallel_write_handle)

# Write "hello" (i.e. 5 bytes) to the data object.
# Notice we didn't specify which stream to use.
curl -H "Authorization: Bearer $bearer_token" "$http_api_url" \
  -F 'op=write'                                              \
  -F "parallel-write-handle=$transfer_handle"                \
  -F 'count=5'                                               \
  -F 'bytes=hello;type=application/octet-stream'             \
  | jq

# Shutdown all streams and update the catalog.
curl -H "Authorization: Bearer $bearer_token" "$http_api_url" \
  --data-urlencode 'op=parallel_write_shutdown'              \
  --data-urlencode "parallel-write-handle=$transfer_handle"   \
  | jq
```

PERFORMANCE

The initial release of the iRODS HTTP API is tested here. These results are preliminary and we expect future releases to be more performant as the API is deployed into more diverse scenarios.

Throughput

This first test harness included two machines, one on a ”home” network, connected over the commodity internet, to another machine at ”work”. The test program is a custom-built, non-optimized Java application which speaks to the v0.1.0 iRODS HTTP API. The test program uses the multi-1247 parallel-write-handle approach mentioned in the section above. In Table 1, this Java application is compared to `iput` which was configured to use multiple high ports for traditional iRODS parallel transfer.

Client	Protocol	Average Elapsed Time (seconds)
iput (high ports)	iRODS	50.113
java application (multi-1247)	HTTP API + iRODS	51.975

Table 1. Upload speed comparison (n=5)

Each application used 4 threads to upload a 100 MiB test file into iRODS 4.3.0 five times and then the elapsed time was averaged. The Java application and HTTP API added a 3.7% overhead compared to `iput`. Additional testing showed that performance is sensitive to buffer sizes and the number of threads used. These results show that comparable performance to `iput` is within reach to many new clients with simple wrappers around the HTTP API.

Requests Per Second

This second test used ApacheBench [10] to measure Requests Per Second (RPS) supported by the HTTP API. Configured to send 2000 total requests, the HTTP API maintained 500 concurrent requests at all times. All of these tests were performed with a single development machine with 32 cores and 256 GiB of RAM. The iRODS server was a custom build of iRODS ~4.3.1 (not yet released) which supported `rc_switch_user` and included updates to the connection pool library. The iRODS HTTP API had optimizations enabled in the build and was configured with 32 threads each for foreground and background processing.

Endpoint	Requests Per Second	50% of requests served (in seconds)
/authenticate	133.20	3.670
/data-objects	697.18	0.686
/resources	2599.53	0.167

Table 2. Requests Per Second (n=2000)

In Table 2, the response times for authenticating a new user using basic/native authentication, reading 8192 bytes of an iRODS data object, and stat'ing a resource are shown. Authentication is the heaviest amount of work and the HTTP API handled 133 requests per second with 50% of requests being served within 3.6 seconds. Reading 8k of data from a data object within iRODS (on a single server) was in the middle with nearly 700 requests per second with 50% of requests being served within two-thirds of a second. Executing a stat operation on a resource was much quicker with nearly 2600 requests per second with 50% of requests being served within one-sixth of a second.

Requests that would traverse to a second or third iRODS server would take more time in a topology test environment. That was not tested here.

EXAMPLES

The following examples show some requests and responses from the iRODS HTTP API. Responses are in JSON and are shown here parsed through the command line tool `jq` for clarity and readability.

stat'ing a collection

```
base_url="http://localhost:9000/irods-http-api/0.1.0"
bearer_token=$(curl -sX POST --user 'rods:rods' "${base_url}/authenticate")

curl -sG -H "Authorization: Bearer $bearer_token" \
  "${base_url}/collections"                       \
  --data-urlencode 'op=stat'                      \
  --data-urlencode 'lpath=/tempZone/home/rods'    \
  | jq

{
  "inheritance_enabled": false,
  "irods_response": {
    "status_code": 0
  },
  "modified_at": 1686499669,
  "permissions": [
    {
      "name": "rods",
      "perm": "own",
      "type": "rodsadmin",
      "zone": "tempZone"
    }
  ],
  "registered": true,
  "type": "collection"
}
```

listing available rule engine plugins

```
base_url="http://localhost:9000/irods-http-api/0.1.0"
bearer_token=$(curl -sX POST --user 'rods:rods' "${base_url}/authenticate")

curl -sG -H "Authorization: Bearer $bearer_token" \
  "${base_url}/rules"                             \
  --data-urlencode 'op=list_rule_engines'          \
  | jq

{
  "irods_response": {
    "status_code": 0
  },
  "rule_engine_plugin_instances": [
    "irods_rule_engine_plugin-irods_rule_language-instance",
    "irods_rule_engine_plugin-cpp_default_policy-instance"
  ]
}
```

REMAINING WORK

This initial release covers basic endpoints and validates the approach for this new API. There are a number of items in the near term that need to be added to make this useful for others. This list includes:

- Implement tests
- Improve performance of /authenticate endpoint
- Consider batch / bulk operations
- Add support for GenQuery2
- Clean up implementation for contributors
- Fill out documentation
- Define good defaults for I/O-specific configuration properties
- Add support for Docker and Docker-Compose
- Expose SSL configuration properties for iRODS communication

FUTURE PLANS

Other considerations we are thinking about, but do not yet have clarity on, include the following possible future directions:

- Consider adding options for enabling/disabling features, endpoints, etc
  - The iRODS C++ REST API supported this for all endpoints
    - Is this the responsibility of the proxy (e.g. nginx, apache httpd)?
- Consider how to best support load balancers
  - Parallel Writes are stateful

- Consider how to deal with long running agents containing stale information
  - Should we refresh the connection after N number of API operations?
  - Should we refresh the connection after certain API operations?
    - e.g. Resource management operations

We are hoping that our approach of providing well-known and well-defined HTTP accessibility to the iRODS Protocol will bring in new developers and new use cases and better expose the flexibility and power of the iRODS server.

REFERENCES

[1] iRODS HTTP API `https://github.com/irods/irods_client_http_api`

[2] Coposky, J., Russell, T. (2021) iRODS Client: C++ REST API. iRODS User Group Meeting 2021. `https://irods.org/uploads/2021/Coposky-iRODS-C_Plus_Plus_REST_API-paper.pdf`

[3] Pistache: An elegant C++ REST Framework. `https://pistacheio.github.io/pistache/`

[4] Russell, T., White, V. (2023) iRODS S3 API: Presenting iRODS as S3. iRODS User Group Meeting 2023. `https://irods.org/uploads/2023/Russell-iRODS-S3-API_Presenting_iRODS_as_S3-paper.pdf`

[5] Draughn, K., Russell, T., King, A. (2023) Can the REST API be improved? `https://gist.github.com/korydraughn/78ec96120234659db1c2ba3235efa46c`

[6] Boost.Beast `https://github.com/boostorg/beast`

[7] Draughn, K., Russell, T., Mieczkowski, A., Coposky, J., Conway, M. (2020) iRODS Client: NFSRODS 1.0. iRODS User Group Meeting 2020. `https://irods.org/uploads/2020/Draughn-iRODS-NFSRODS_v1.0.0-paper.pdf`

[8] PR #7047. Client connection libraries allow deferring/changing the auth logic `https://github.com/irods/irods/pull/7047`

[9] PR #6691. Implemented new API plugin: rc_switch_user `https://github.com/irods/irods/pull/6691`

[10] ApacheBench `https://httpd.apache.org/docs/2.4/programs/ab.html`

An update on Yoda: Using iRODS to manage data throughout your research

Lazlo Westerhof
Utrecht University
l.r.westerhof@uu.nl

ABSTRACT

The landscape of research data management can present challenges to researchers seeking to manage, share, and publish their work. Since 2014, Utrecht University has been addressing these challenges through the development of Yoda, a research data management system designed to facilitate researchers to securely deposit, describe, share, publish and preserve large amounts of research data in compliance with the FAIR principles during all phases of a research project.

Yoda was previously presented at the iRODS user group meeting in 2018. Since then, it has undergone significant development regarding workflows, metadata editing, asynchronous processes, plugins, and APIs. Yoda has been continuously improved and has been deployed in our institutes for several years. It is currently used by more than 10,000 researchers and students and manages over 3 petabytes of data. Additionally, Yoda is publicly available as open-source software with a permissive license.

This session will explore the evolution and progression of Yoda and its continued integration with the iRODS platform over the past five years. We will discuss Yoda design principles, new features and how they are implemented in iRODS.

The iRODS CLI we deserve

Derek Dong
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
ddong@renci.org

Kory Draughn
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
korydraughn@renci.org

Terrell Russell
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
unc@terrellrussell.com

ABSTRACT

The current iRODS iCommands are a culmination of many years of effort, but they are beginning to show their age, especially in terms of design and extensibility. We aim to create a brand new CLI that focuses on using modern libraries (iRODS or otherwise), modern C++, being extensible and modular, and provide a single binary, rather than ~50. This talk will cover the current plans and progress towards this effort.

GoCommands: A cross-platform command-line client for iRODS

Illyoung Choi
CyVerse / University of Arizona
iychoi@arizona.edu

Edwin Skidmore
CyVerse / University of Arizona
edwins@arizona.edu

Nirav Merchant
CyVerse / University of Arizona
nirav@arizona.edu

ABSTRACT

The diversity of scientific computing platforms has increased significantly, ranging from small devices like Raspberry Pi to large computing clusters. However, accessing iRODS data on these varied platforms remains a common but challenging requirement. The official command-line tool for iRODS, iCommands, is limited to a few platforms like CentOS7 and Ubuntu 18/20. As a result, users on other platforms like MacOS, Windows, and Raspberry Pi OS have no straightforward performant means of accessing iRODS.

GoCommands is another command-line tool for iRODS designed to address the portability issue of iCommands. Written in Go programming language, building its executable for diverse platforms is straightforward. The tool is a single executable that does not require any dependency installation. Pre-built binaries for MacOS, Linux (any distros), and Windows, regardless of their CPU architectures, are already available. In addition, the tool does not require elevated privileges for installation and run. This makes it possible for users on nearly any platform to access iRODS.

One of the noteworthy new commands introduced in GoCommands is 'bput', which enables efficient uploading of many small files. GoCommands also includes a reimplementation of 'put', 'get', and 'sync' commands in iCommands. By default, GoCommands transfers data in parallel, which greatly improves the performance of accessing iRODS from various platforms. GoCommands showed 127MBps for upload and 134MBps for download in CyVerse Discovery Environment when accessing CyVerse Data Store.

Additionally, GoCommands is capable of working with iCommands' configuration file. We will be providing a demo on how to install, configure, and use GoCommands.

GoCommands is currently deployed to several research projects for managing data. In this talk, we will be presenting how MagAO-X astronomy project and Open Forest Observatory project manage data using GoCommands.

We expect the new tool, GoCommands, will allow researchers utilizing diverse computational platforms to readily use it for managing their data.

iRODS UGM 2023, June 13-16, 2023, Chapel Hill, NC

Authentication in iRODS 4.3: Investigating OAuth2 and OpenID Connect (OIDC)

Martin Flores
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
martinflores@renci.org

ABSTRACT

This talk will provide an overview and demonstration of exploratory work with OAuth 2.0, OpenID Connect, and the new iRODS HTTP API. A successful proof of concept will show the community how iRODS integrations with other authentication services may be best handled in the future. Feedback and insights are welcome.

RSpace + iRODS: Update: Plans and Opportunities

Rory Macneil
Research Space
rmacneil@researchspace.com

Terrell Russell
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
unc@terrellrussell.com

ABSTRACT

This presentation will include an overview and brief history of the RSpace + iRODS integration, a description of the next phase of development to expose metadata in RSpace to iRODS, and an outline of the vision of RSpace + iRODS as a unifying element in Research Commons and other research infrastructures. The last part will use Digital Research Alliance of Canada's Research Commons and EOSC's EUDAT Collaborative Data Infrastructure as examples.

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

- 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

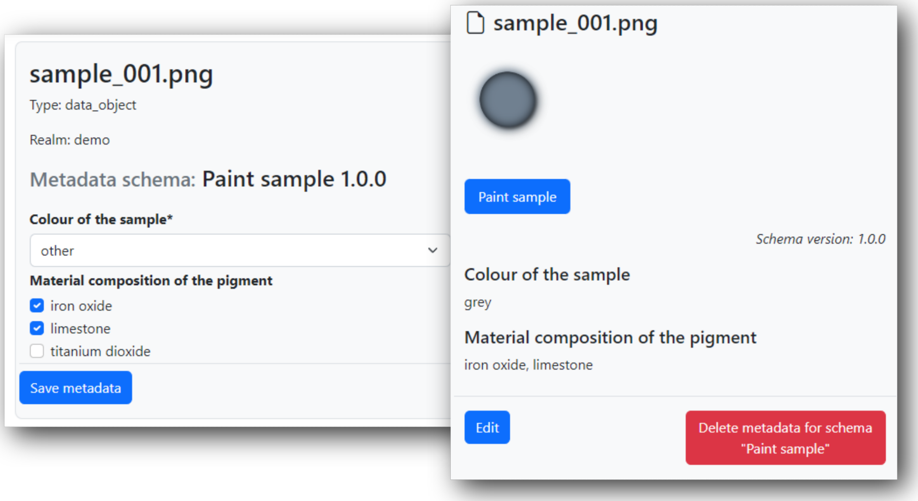
In the broadest sense of the term, metadata is data about data, i.e. description of data that can be embedded inside a file, attached to it as a tag or contained in an additional document (*aka* sidecar file). In iRODS, it takes the form of AVUs (attribute-value-unit tuples) stored in the iCAT database [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).

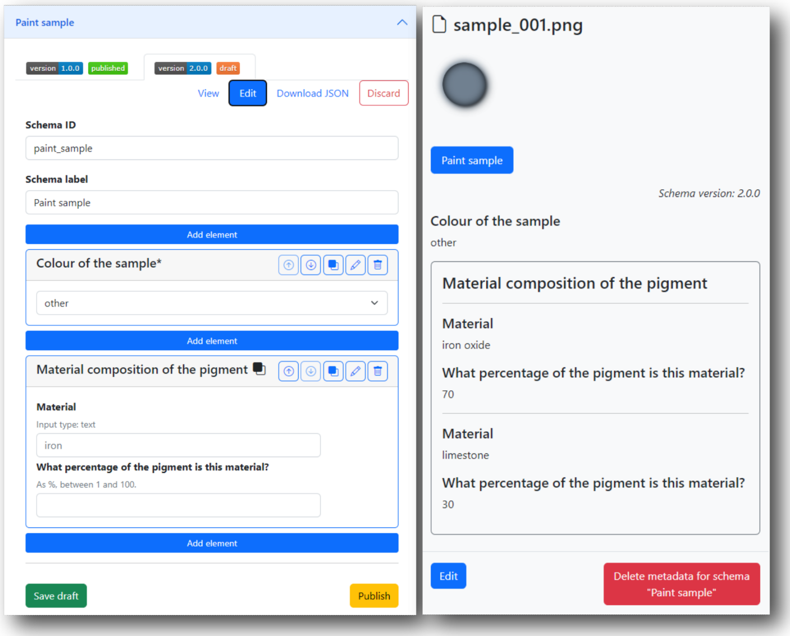
Concretely, when a metadata schema is first created, it is mostly saved as a draft with version 1.0.0. This draft can be edited and even deleted, but it is not available for applying metadata, since it is unstable and thus unreliable. In order to use the schema for applying metadata, it must be published. 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.



Schema ID
paint_sample

Schema label
Paint sample

Colour of the sample*
other

Material composition of the pigment

Material
Input type: text
iron

What percentage of the pigment is this material?
As %, between 1 and 100.

sample_001.png

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

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.

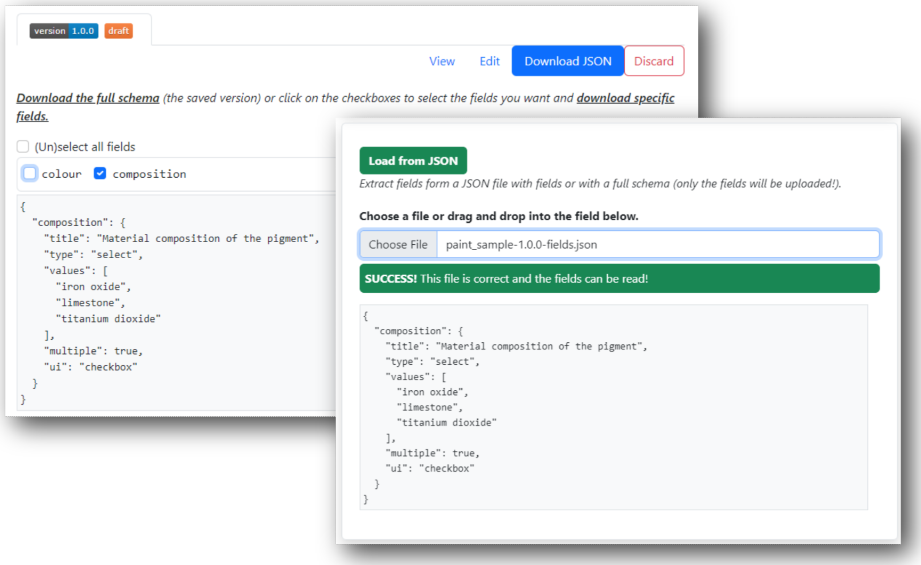
Finally, because composite fields are like miniature schemas, their representation combines features of both fields and schemas. Next to the compulsory `type` (which must be 'object') and `title` and the optional `help` and `repeatable`, they must include a `properties` attribute that follows the same 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.

Choose File 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

Integrating iRODS with Project Eureka and Open OnDemand

Boyd Wilson
Omnibond
boyd@omnibond.com

David Reynolds
Omnibond
david@omnibond.com

ABSTRACT

This talk will discuss how we are integrating iRODS into our next evolution of CloudyCluster called Project Eureka. Part of Eureka is a project-based interface in Open OnDemand which will include a storage management UI built to work directly with iRODS. We will show a demonstration of the work in progress and request feedback from the community.

iRODS UGM 2023, June 13-16, 2023, Chapel Hill, NC

rirods: An R client for iRODS

Martin Schobben
Vienna University of Technology
schobbenmartin@gmail.com

Mariana Montes
KU Leuven
mariana.montes@kuleuven.be

Christine Staiger
Utrecht University
c.staiger@uu.nl

Terrell Russell
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
unc@terrellrussell.com

ABSTRACT

In this talk we present a new client for iRODS: the R package rirods. In contrast to its predecessor (Chytracek et al. 2015), this package is pure R (rather than C++) and transfers data over HTTP, communicating with the iRODS REST API.

R is a very popular language in data science, and we expect that many R users who are not familiar with Python, or the command line, will benefit from interacting with iRODS through this R package. We will showcase the main functionalities of the package and what we have planned for the future. Crucially, we offer the equivalents of iCommands 'iput', 'iget', 'ils', 'imeta' and a few other functions, but also rirods-specific functions that allow the user to directly stream between memory and iRODS without staging files locally.

iRODS S3 API: Presenting iRODS as S3

Terrell Russell
Renaissance Computing
Institute (RENCI)
UNC Chapel Hill
unc@terrellrussell.com

Violet White
Renaissance Computing
Institute (RENCI)
UNC Chapel Hill
vem@renci.org

ABSTRACT

S3 has taken over the storage world for a number of good reasons. Many software libraries, tools, and applications now read and write the S3 protocol directly. This paper describes a new iRODS client API that presents the iRODS namespace as S3. It will discuss the requirements, the design, the initial implementation, and future work.

Keywords

iRODS, s3, api, cpp

DESIRE / JUSTIFICATION

The iRODS Protocol has been around for more than 20 years and has proven reliable and robust. However, it is also relatively complex for users and developers who only want to move data. Amazon's introduction of the Simple Storage Service (S3) [1] in 2006 defined the S3 protocol on top of HTTP(S). Its relatively limited scope of moving and labeling data, while providing online access and near-perfect uptime, resiliency, and availability alongside other new Amazon Web Services (AWS) [2], came to dominate the online storage provider space. Many other tools grew the ability to interact with S3 and now new tooling is expected to be able to do the same upon initial release. In addition to being relatively comprehensible and easy to implement as a client, it also decouples authentication from authorization.

Implementing a new iRODS S3 API [3] would align with our Protocol Plumbing [4] efforts from the last couple years and will make iRODS more accessible and approachable to new developers as well as provide the iRODS Consortium a maintainable, familiar front door for new partners.

Because of this, the iRODS Consortium decided to implement such an API and present iRODS as S3. Our goals were to reuse as much code as possible (from other projects that we have written AND from other projects others have written), to provide a solution that is friendly to be deployed in a high availability (HA) environment, and to be maintainable for future developers.

Possible Futures

In July of 2021, an initial charter was sent to the iRODS Community, beginning the work of the newly designated iRODS S3 Working Group [5]. In it, four options were introduced as candidates for how to present iRODS as S3:

1. Update and maintain the MinIO-iRODS Gateway from BioTeam [6]

   This MinIO-based S3 front-end already existed and had been demonstrated, but had not been updated since its debut in 2018. It was never deployed in production and served only as a proof of concept. It was built with the GoRODS client library [7], which has been archived earlier this year. Presumably, BioTeam would have continued to own/maintain GoRODS and the MinIO-iRODS-gateway if this option was selected.

iRODS UGM 2023 June 13-16, 2023, Chapel Hill, NC

2. MinIO-iRODS-Gateway is converted to use the Go iRODS Client Library [8]

   Illyoung Choi, at the CyVerse project at the University of Arizona, has produced and is actively developing a pure Go iRODS client library. This new library could be ”swapped” for the GoRODS calls in the MinIO-iRODS-gateway. Presumably, then Arizona / Illyoung would own/maintain a fork of the MinIO-iRODS-gateway.

3. Add irods/gateway-irods.go to upstream MinIO project [9]

   Someone in the community (perhaps the Consortium itself) would port or implement the same work from Option 2 (convert to pure Go), but work with the MinIO community to get iRODS to be an officially supported gateway for the MinIO server directly.

4. New C++ implementation [3]

   The iRODS Consortium would work to implement the S3 specification directly with a new C++ client. This could be more performant (in the long run), but requires the most work and will require answers to many open questions.

At the end of this charter, Terrell Russell wrote ”I am leaning towards Option 3 as the best option both from a cost/benefit perspective, as well as exposure to a larger community and confidence that ’it just works’.”

Research

The Consortium investigated these four options and began implementation across four phases over the following two years. Phase 1 covered and discarded the first three options relatively quickly. Phase 2 selected and explored a C++ proof of concept while also investigating many alternative possibilities. Phases 3 and 4 explored multipart data transfer and authentication.

Phase 1

In July of 2021, Option 1’s MinIO-iRODS-Gateway with GoRODS wrapping the C library was deemed limited and not as maintainable by either BioTeam or by the Consortium.

Later, the MinIO-iRODS-Gateway with pure go-irodsclient of Option 2 was determined to require an anonymous iRODS ticket functionality to handle the requests. The Consortium investigated and implemented a TicketBooth (or BoxOffice) application in October 2021 to handle creation and dispersal of these anonymous tickets. However, by November 2021 it was clear that such an application would require rodsadmin credentials making the TicketBooth no more functional than just using the C++ REST API directly. In addition, by February 2022, the use case of independently serving multiple users’ files was proving impossible because the ’gateway’ code fires ’too late’ in the MinIO application to discriminate between different incoming users. ’Who’ was logging in had already been determined by the MinIO server core.

In May 2022 [10], all of this design and implementation effort became moot when MinIO announced the deprecation of the gateway altogether. Their announcement explained that it was no longer worth it to support ’legacy POSIX-based’ systems. Needless to say, this removed MinIO as a viable option.

Phase 2

With the first three options off the table, the Consortium leaned into a new C++ implementation of an S3 API. This implementation would remove dependencies on other codebase(s) and companies and business models. This implementation would need to support multi-user *and* multi-bucket use cases. The Consortium began evaluation and selection of a C++ framework for handling the networking and programming interfaces in August of 2022. The list of candidates included Pistache [11], Oat++ [12], Drogon [13], and Boost.Beast [14]. Boost.Beast was selected in November 2022 and some initial endpoints were functional and responding by January of 2023, including configuration for both user mappings (from keypairs to iRODS users) and bucket mappings (from bucket names to iRODS collections).

Phase 2 - Alternate Illyoung Universes

While the C++ implementation was first coming together, Illyoung Choi at the University of Arizona and the CyVerse project was busy exploring additional alternate possibilities that would prevent the Consortium from having to build and support a novel codebase. The following were considered but eventually all discarded because they could not satisfy the dual requirement for multi-user and multi-bucket support:

- Add S3 protocol support to SFTPGo [16] (August 2022)
- Searching for existing S3 server in Go (September 2022)
- Add iRODS backend to Zenko [17] (October 2022)
- Add JuiceFS [18] frontend to iRODS (November 2022)
- Add JuiceFS frontend to SFTPGo (November 2022)
- GarageHQ [19] frontend to iRODS (January 2023)
- Using in-memory IBM s3mem-go [20] as inspiration (March 2023)

Phase 3

Once the decision was made for a novel C++ implementation of an S3 API, and the initial endpoints were coming together relatively quickly, attention was turned to moving data from a client into the API efficiently. This means addressing the Multipart Upload endpoint.

There were four options that the Consortium could see for implementing the S3 protocol's multipart upload. These were named 'Multiobject', 'Store-and-forward', 'Efficient Store-and-forward', and 'Store-and-register'.

The first of these, Multiobject, would write all uploaded parts individually to iRODS, and then a final step or 'complete' would trigger copy or concatenate or some other function in the server to stitch all the parts together as a single file in the iRODS namespace. The main benefit to this approach is that it is relatively simple and straightforward. However, the downsides are that a lot of additional policy would fire on the server (opens, reads, writes, closes for every part) which may trigger replication to distant, expensive disks (or continents). It also would require a new API plugin or function in the server to 'concatenate()' the individual parts.

The second option was Store-and-forward. This approach would write all the parts to storage local to the API, as a single file, and then send that entire file to iRODS where it would be written into managed storage. This is also very simple to implement, and does not fire additional policy in the server (the file is only written once in iRODS). However, since the file has to be sent 'twice', the client will 'see' a slow or delayed experience. Additionally, the API itself would have to provide potentially an enormous disk, realistically large enough to support the largest incoming file multiplied by the number of concurrent users.

A third option would be an opportunistic Efficient Store-and-forward algorithm that has a fallback of regular Store-and-forward. If the parts that are being uploaded are known to be contiguous, then they can be sent along to iRODS while any non-contiguous parts can be held in memory (or disk) in the API until enough parts have arrived that they too can be considered contiguous and eligible for sending to iRODS. This is elegant and would only require a single write to iRODS but has more complexity in the code since there are more states to be managed. This approach would also still require a relatively large disk for scratch space since if an early part was not delivered, no additional parts could be sent to iRODS until the earlier part had arrived. It is also noted here that the S3 protocol reserves the right to send a part number more than once which means that the S3 API would have to be able to recover in that scenario. This might only be possible if a resent part number is guaranteed to be exactly the same size as the initial part sent with that part number. If this is not the case, then no efficiency can be gained by this option over naive Store-and-forward.

The fourth option brainstormed was Store-and-register. In this scenario, the S3 API would write all the parts into their final order in storage that is also visible to iRODS. Then, the S3 API would issue a 'register' operation to have the file added as a replica to the iRODS catalog in-place. This is the simplest and the fastest but would only trigger the 'register' policy on the server. And, in addition, this adds a significant assumption and/or requirement of co-visibility of the S3 API's storage to the iRODS server.

These four options all have their upsides and their downsides and it was not clear which approach should be implemented first. It was decided that an initial release would not include multipart upload as more research needed to be done.

Phase 4

In June of 2023, with most of the endpoints implemented (see the next section), and multipart uploads set aside for a later date, there are now tests written and passing with the Python boto3 [15] client. However, more work needs to be done with the HMAC signature matching for other clients (AWS cli, MinIO's mc, etc.)

STATUS / IMPLEMENTATION

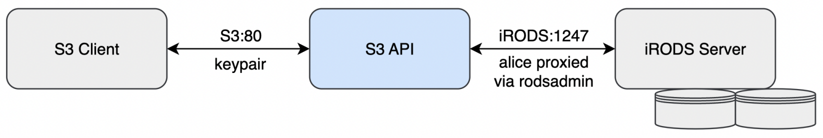


Figure 1. Architecture of the iRODS S3 API

The first implementation of the iRODS S3 API is a single binary currently requiring rodsadmin credentials (so it can proxy as the authenticated user). It has two configuration files, currently `irods_environment.json`, to define the iRODS environment and connection information, and `config.json`, to control the S3 features. This will probably be consolidated into a single configuration file as the settings mature and settle.

Architecture

The implemented endpoints at the time of this writing include:

- CopyObject
- DeleteObject
- GetObject
- HeadObject
- ListObjectsV2
- PutObject

The multipart endpoints are still under discussion, but include:

- CompleteMultipartUpload

- CreateMultipartUpload
- UploadPart

The Consortium has not yet looked into the following endpoints. They may be implemented later depending on community feedback and/or existing client implementations that require these endpoints:

- UploadPartCopy
- ListObjects
- DeleteObjects
- GetObjectAcl
- PutObjectAcl
- GetObjectTagging
- PutObjectTagging

Configuration

As stated, the configuration file for the S3 API will be consolidated and have 'two sides', one for the `s3_server`, and one for the `irods_client`.

```
{
    // Defines S3 options that affect how the
    // client-facing component of the server behaves.
    "s3_server": {
        // ...
    },

    // Defines iRODS connection information.
    "irods_client": {
        // ...
    }
}
```

The `s3_server` side will contain host and port information, control for logging level and the mapping system for both users and buckets. Initial designs for the mapping system will include a static resolver for both users and buckets, directly in this file, however, later this information could come from an external source and provide dynamic resolution for production deployments.

```
"s3_server": {
    "host": "0.0.0.0",
    "port": 80,

    "log_level": "warn",

    "bucket_mapping": {
        # local / static (JSON)
        # external / dynamic - iRODS or third party API
    },

    "user_mapping": {
        # local / static (JSON)
        # external / dynamic - iRODS or third party API
    },

    "threads": 3
}
```

The `irods_client` side contains the standard iRODS connection information including the host, port, and zone name, as well as the rodsadmin account information and some connection pooling information for efficiency when communicating with the iRODS server.

```
"irods_client": {
    "host": "<string>",
    "port": 1247,
    "zone": "<zone>",

    "proxy_rodsadmin": {
        "username": "<string>",
        "password": "<string>"
    },

    "connection_pool": {
        "size": 6,
        "refresh_timeout_in_seconds": 600
    }
}
```

NEXT STEPS

The next steps for this project include configuration consolidation and HMAC cleanup and confirmation. An initial release will require packaging as well as potential deployment instructions. Before that happens, testing will need to be done with multiple clients, perhaps in parallel, and even as an iRODS S3 resource. And of course, additional mappings for both user and bucket will be explored to see what satisfies the community's needs and expectations.

CONCLUSION

This new iRODS S3 API is exciting and will open the iRODS policy ecosystem to numerous additional existing S3 clients. Multiple systems will be able to take advantage of an iRODS Zone without needing to write code or learn the iRODS protocol. This new API is a key part of the Consortium's efforts around Protocol Plumbing.

REFERENCES

[1] AWS S3. `https://aws.amazon.com/s3`

[2] Amazon Web Services. `https://aws.amazon.com`

[3] iRODS S3 API. `https://github.com/irods/irods_client_s3_api`

[4] Russell, T. (2022) Protocol Plumbing: Presenting iRODS as WebDAV, FUSE, REST, NFS, SFTP, K8s CSI, and S3. Supercomputing 2022. `https://slides.com/irods/sc22-irods-protocol-plumbing`

[5] iRODS S3 Working Group. https://github.com/irods-contrib/irods_working_group_s3
[6] https://github.com/bioteam/minio-irods-gateway
[7] https://github.com/jacquayj/GoRODS
[8] https://github.com/cyverse/go-irodsclient
[9] https://github.com/minio/minio/tree/master/cmd/gateway
[10] https://blog.min.io/deprecation-of-the-minio-gateway/
[11] https://github.com/pistacheio/pistache
[12] https://oatpp.io/
[13] https://drogon.org/
[14] https://github.com/boostorg/beast
[15] https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
[16] https://github.com/drakkan/sftpgo
[17] https://github.com/scality/Zenko
[18] https://juicefs.com/en/
[19] https://garagehq.deuxfleurs.fr/
[20] https://github.com/IBM/s3mem-go

Using iRODS Rules to Automate Trash Management Policy

Urvika Gola
CyVerse / University of Arizona
ugola@arizona.edu

ABSTRACT

Effective trash removal policies are essential for data storage and management. This presentation will share a solution that harnesses microservices and dynamic policy enforcement points (PEPs) in iRODS for efficient, automated trash management for both data objects and collections implemented as rule logic. Data can be put into trash through a variety of methods, and dynamic Policy Enforcement Points (PEPs) offer the flexibility to make informed policy decisions for each approach. By constructing four distinct policy enforcement points - pre, post, except, and finally varieties, we ensure that our trash management system adeptly handles various data movement techniques, ensuring optimal efficiency. We will provide an in-depth exploration of the dynamic PEPs and microservices used in our solution, elaborating on their implementation, advantages, and challenges we have overcome. At the end of this presentation, our goal is to provide attendees with insights into the power of iRODS microservices and dynamic PEPs, enabling them to leverage this knowledge for streamlining their own data management needs.

iRODS UGM 2023, June 13-16, 2023, Chapel Hill, NC

Updates on iRODS Data Repository Service Adapter

Mike Conway
NIEHS / NIH
mike.conway@nih.gov

Deep Patel
NIEHS / NIH
deep.patel@nih.gov

ABSTRACT

The GA4GH Data Repository Service (DRS) standard is part of a family of standards for distributed, federated data analysis. Using standard workflow languages such as WDL, CWL, and Nextflow, these standards allow workflows to dispatch containerized tasks to run at appropriate locations, including across cloud providers and on-prem compute environments. The DRS standard provides an abstraction over distributed data sources, allowing these workflow tasks to authorize data access and access underlying data sets.

A DRS implementation over iRODS allows the iRODS data grid to expose data to this federated analysis ecosystem. The Federated Analysis System Project (FASP) components represent a formalization of the iRODS 'compute to data' pattern for the important Genomics and Health community.

iRODS UGM 2023, June 13-16, 2023, Chapel Hill, NC

GenQuery2: A more standardized, powerful parser for the iRODS namespace

Kory Draughn
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
korydraughn@renci.org

Terrell Russell
Renaissance Computing Institute (RENCI)
UNC Chapel Hill
unc@terrellrussell.com

ABSTRACT

The iRODS GenQuery interface has long defined the way users and administrators can search the iRODS namespace, its storage systems, users, and metadata, while honoring the iRODS permission model. The next generation of GenQuery, GenQuery2, is now available for experimentation. However, there is still a lot of work to do. This paper will cover its expanded syntax and capabilities and what is to come.

Keywords

iRODS, general query, genquery, parser, SQL

PURPOSE

GenQuery2 is an experimental redesign (and implementation) of the iRODS GenQuery parser. Written in flex and bison, this new implementation provides full and clear control over the domain-specific language (DSL) that is used to power many of the internal processes within iRODS.

This project exists as a means for allowing the iRODS community to test the implementation and provide feedback so that the iRODS Consortium can produce a GenQuery parser that is easy to understand, maintain, and enhance all while providing a syntax that mirrors standard SQL as much as possible.

Once stable, the code will be merged into the iRODS server making it available within and alongside future releases of iRODS.

The current source can be found at: `https://github.com/irods/irods_api_plugin_genquery2`

The repository contains all source code for generating a package which includes the following three binaries: an API plugin, a rule engine plugin, and a new iCommand. Everything discussed in this paper can be found in the repository.

GENERAL FEATURES

The features of this new parser are listed here. Many of these are new or correctly supported in iRODS for the first time. They represent the vast majority of requested new features and bug fixes for GenQuery1 from the iRODS Community over the last decade and provide a flexible and powerful new way to interact with the iRODS Catalog.

- Enforces the iRODS permission model
- Logical AND, OR, and NOT
- Grouping via parentheses

iRODS UGM 2023 June 13-16, 2023, Chapel Hill, NC

- SQL CAST
- SQL GROUP BY
- SQL aggregate functions (e.g. count, sum, avg, etc)
- Per-column sorting via ORDER BY [ASC|DESC]
- SQL FETCH FIRST N ROWS ONLY (LIMIT offered as an alias)
- Metadata queries involving different iRODS entities (i.e. data objects, collections, users, and resources)
- Operators: =, !=, <, <=, >, >=, LIKE, BETWEEN, IS [NOT] NULL
- SQL keywords are case-insensitive
- Federation is supported and backwards compatible

COMPONENTS AND EXAMPLES

To provide this new parser to the different parts of iRODS, three components have been designed and implemented. These include an API plugin, a rule engine plugin, and a new iCommand, namely **iquery**.

API Plugin

The API Plugin wraps the parser and makes it available to all iRODS clients. It has a new API number of **1000001**, which may change in the future, and the following input parameters:

- `query_string` - The GenQuery2 string.
- `zone` - The name of the zone where the query should be executed.
- `sql_only` - An integer instructing the plugin to return the generated SQL, without execution.

This API Plugin returns a JSON string representing the `resultset` on success. On failure, it returns an iRODS error code.

At the moment, this API Plugin defaults to returning a maximum of 16 rows if the client does not specify the number of rows to return. This has not yet been tested heavily and will probably change with additional community input.

Rule Engine Plugin

The provided Rule Engine Plugin makes GenQuery2 available to the iRODS Rule Language and other rule engine plugins via four rules of its own:

- `genquery2_execute(*handle, *query_string)`
- `genquery2_next_row(*handle)`
- `genquery2_column(*handle, *index, *value)`
- `genquery2_destroy(*handle)`

These rules can be called in succession to execute a query, walk the results, and then clean up the memory allocations.

Example

The rules can be enabled by adding the following rule engine instance information to the `rule_engines` stanza of `server_config.json`.

```
{
    "instance_name": "irods_rule_engine-genquery2-instance",
    "plugin_name": "irods_rule_engine-genquery2",
    "plugin_specific_configuration": {}
}
```

Then, an example rule could look like the following:

```
genquery2_test_rule()
{
    # Execute a query. The results are stored in the Rule Engine Plugin.
    genquery2_execute(*handle, "select COLL_NAME, DATA_NAME order by DATA_NAME desc limit 1");

    # Iterate over the resutls.
    while (errorcode(genquery2_next_row(*handle)) == 0) {
        genquery2_column(*handle, '0', *coll_name); # Copy the COLL_NAME into *coll_name.
        genquery2_column(*handle, '1', *data_name); # Copy the DATA_NAME into *data_name.
        writeLine("stdout", "logical path => [*coll_name/*data_name]");
    }

    # Free any allocated resources. This is also handled automatically when the agent is shut down.
    genquery2_destroy(*handle);
}
```

This rule selects a single row of `COLL_NAME` and `DATA_NAME`, ordered by descending `DATA_NAME`, and then writes a constructed full logical path to `stdout`.

iCommand

The last component provided by this project is a new iCommand client which uses the new API Plugin. This new binary, `iquery`, enables execution of GenQuery2 queries against the iRODS Catalog from the command line.

The current help text:

```
Usage: iquery [OPTION]... QUERY_STRING

Queries the iRODS Catalog using GenQuery2.

QUERY_STRING is expected to be a string matching the GenQuery2 syntax. Failing
to meet this requirement will result in an error.

Mandatory arguments to long options are mandatory for short options too.

Options:
      --sql-only        Print the SQL generated by the parser. The generated
```

```
                        SQL will not be executed.
  -z, --zone=ZONE_NAME  The name of the zone to run the query against. Defaults
                        to the local zone.
  -h, --help            Display this help message and exit.

iRODS Version 4.3.0                iquery (experimental)
```

Examples

To list the number of replicas for all data objects, the following `iquery` command can be run. `jq` is used to cleanly format the resulting JSON received on `stdout`. There are two rows returned, with three columns each. These represent the collection name, data object name, and the count of how many times the compound grouping was found, in this case, 3 and 1 times, respectively.

```
$ iquery "select COLL_NAME, DATA_NAME, count(DATA_ID) group by COLL_NAME, DATA_NAME" | jq
[
  [
    "/tempZone/home/rods",
    "foo",
    "3"
  ],
  [
    "/tempZone/home/rods",
    "bar",
    "1"
  ]
]
```

The next example shows the SQL generated by the parser for the same query, but without executing that SQL. This time, `pg_format` is used to format the returned `stdout`.

```
$ iquery --sql-only \
     "select COLL_NAME, DATA_NAME, count(DATA_ID) group by COLL_NAME, DATA_NAME" | \
     pg_format -

SELECT DISTINCT
    t0.coll_name,
    t1.data_name,
    count(t1.data_id)
FROM
    R_COLL_MAIN t0
    INNER JOIN R_DATA_MAIN t1 ON t0.coll_id = t1.coll_id
    INNER JOIN R_OBJT_ACCESS pdoa ON t1.data_id = pdoa.object_id
    INNER JOIN R_TOKN_MAIN pdt ON pdoa.access_type_id = pdt.token_id
    INNER JOIN R_USER_MAIN pdu ON pdoa.user_id = pdu.user_id
    INNER JOIN R_OBJT_ACCESS pcoa ON t0.coll_id = pcoa.object_id
    INNER JOIN R_TOKN_MAIN pct ON pcoa.access_type_id = pct.token_id
    INNER JOIN R_USER_MAIN pcu ON pcoa.user_id = pcu.user_id
WHERE
    pdu.user_name = ?
    AND pcu.user_name = ?
    AND pdoa.access_type_id >= 1050
    AND pcoa.access_type_id >= 1050
GROUP BY
    t0.coll_name,
    t1.data_name FETCH FIRST 16 ROWS ONLY
```

REMAINING WORK

This project is relatively complete and fulfills many of the desired features, but there are a few items that must be resolved before making GenQuery2 available as part of the default iRODS server. The first is working to clean up the `CMakeLists.txt` file to be more in line with our recent efforts to modernize and modularize our approach to CMake across all our repositories. Second, this repository needs many tests to be considered ready for release and any kind of production use. Third, there is an open discussion about how a new parser should handle interacting with groups (and their constituent users) and tickets (and their permission model).

FUTURE PLANS

The future is GenQuery2, but we are not there yet. There are additional features that need to be considered and carefully implemented to follow the rest of the iRODS permission model, architecture, and design decisions. This list represents a snapshot in time in June 2023:

- Expose more SQL features
  - CASE, HAVING clauses
  - Sub-selects
  - Multi-argument functions
- Consider controlling various options through GenQuery2 syntax
  - e.g. iquery "option distinct off; select DATA_NAME"
- Consider switching from `boost::variant` to `std::variant`
- Simplify pagination
  - Provide a utility library that manages the page information
  - Provide a document explaining how the utility may be implemented

COMMUNITY ENGAGEMENT

At this time, we are considering the idea of releasing GenQuery2 as an experimental package to be installable alongside an existing iRODS instance. This approach allows the community to try GenQuery2 and provide feedback and allows frequent updates that are not tied to a particular server release. We are seeking input and testing and look forward to learning what additional use cases this new parser can solve.