Table of Contents

Chapter 1. Overview

https://github.com/ga4gh/data-repository-service-schemas

1.1. Version information

Version : 0.1.0

1.2. Contact information

Contact : GA4GH Cloud Work Stream
Contact Email : ga4gh-cloud@ga4gh.org

1.3. License information

License : Apache 2.0
License URL : https://raw.githubusercontent.com/ga4gh/data-repository-service-schemas/master/LICENSE
Terms of service : null

1.4. URI scheme

BasePath : /ga4gh/drs/v1
Schemes : HTTPS, HTTP

1.5. Consumes

- `application/json`

1.6. Produces

- `application/json`

Chapter 2. Introduction

The Data Repository Service (DRS) API provides a generic interface to data repositories so data consumers, including workflow systems, can access data in a single, standard way regardless of where it’s stored and how it’s managed. This document describes the DRS API and provides details on the specific endpoints, request formats, and response. It is intended for developers of DRS-compatible services and of clients that will call these DRS services.

The primary functionality of DRS is to map a logical ID to a means for physically retrieving the data represented by the ID. The sections below describe the characteristics of those IDs, the types of data supported, and how the mapping works.

Chapter 3. DRS API Principles

3.1. DRS IDs

Each implementation of DRS can choose its own id scheme, as long as it follows these guidelines:

- DRS IDs are URL-safe text strings made up of alphanumeric characters and any of [.-_/]
- One DRS ID MUST always return the same object data (or, in the case of a collection, the same set of objects). This constraint aids with reproducibility.
- DRS does NOT support semantics around multiple versions of an object. (For example, there’s no notion of “get latest version” or “list all versions” in DRS v1.) Individual implementation MAY choose an ID scheme that includes version hints.
- DRS implementations MAY have more than one ID that maps to the same object.

3.2. DRS Datatypes

DRS v1 supports two datatypes:

- Blobs — these are file-like objects
- Collections — these are sets of other DRS objects (either Blobs or Collections)

3.3. Read-only

DRS v1 is a read-only API. We expect that each implementation will define its own mechanisms and interfaces (graphical and/or programmatic) for adding and updating data.

3.4. URI convention (WORK IN PROGRESS)

For convenience, we define a recommended syntax for fully referencing DRS-accessible objects. Strings of the form drs://<server>/<id> mean “make a DRS call to the HTTP address at <server>, passing in the DRS id <id>, to retrieve the object”. For example, these strings are useful when passing objects to a WES server for processing.

3.5. Standards

The DRS API specification is written in OpenAPI and embodies a RESTful service philosophy. It uses JSON in requests and responses and standard HTTP/HTTPS for information transport.

Chapter 4. Authorization & Authentication

4.1. Making DRS Requests

The DRS implementation is responsible for defining and enforcing an authorization policy that determines which users are allowed to make which requests. We recommend that DRS implementations use an OAuth2 bearer token, although they can choose other mechanisms if appropriate. DRS callers can use the `auth_instructions_url` from the service-info endpoint to learn how to obtain and use a bearer token for a particular implementation.

4.2. Fetching DRS Objects

The DRS API allows implementers to support a variety of different content access policies, depending on what `AccessMethod` s they return:

- public content:
    - server provides an `access_url` with a `url` and no `headers`
    - caller fetches the object bytes without providing any auth info
- private content that requires the caller to have out-of-band auth knowledge (e.g. service account credentials):
    - server provides an `access_url` with a `url` and no `headers`
    - caller fetches the object bytes, passing the auth info they obtained out-of-band
- private content that requires the caller to pass an Authorization token:
    - server provides an `access_url` with a `url` and `headers`
    - caller fetches the object bytes, passing auth info via the specified header(s)
- private content that uses an expensive-to-generate auth mechanism (e.g. a signed URL):
    - server provides an `access_id`
    - caller passes the `access_id` to the `/access` endpoint
    - server provides an `access_url` with the generated mechanism (e.g. a signed URL in the `url` field)
    - caller fetches the object bytes from the `url` (passing auth info from the specified headers, if any)

Chapter 5. Paths

5.1. Retrieve a Data Bundle

```
GET /bundles/{bundle_id}
```

5.1.1. Parameters

Type	Name	Schema
Path	**bundle_id** *required*	string

5.1.2. Responses

HTTP Code	Description	Schema
200	Successfully found the Data Bundle.	GetBundleResponse
400	The request is malformed.	ErrorResponse
401	The request is unauthorized.	ErrorResponse
403	The requester is not authorized to perform this action.	ErrorResponse
404	The requested Data Bundle wasn’t found.	ErrorResponse
500	An unexpected error occurred.	ErrorResponse

5.1.3. Tags

- DataRepositoryService

5.2. Get info about a Data Object.

```
GET /objects/{object_id}
```

5.2.1. Description

Returns object metadata, and a list of access methods that can be used to fetch object bytes.

5.2.2. Parameters

Type	Name	Schema
Path	**object_id** *required*	string

5.2.3. Responses

HTTP Code	Description	Schema
200	The Data Object was found successfully.	GetObjectResponse
400	The request is malformed.	ErrorResponse
401	The request is unauthorized.	ErrorResponse
403	The requester is not authorized to perform this action.	ErrorResponse
404	The requested Data Object wasn't found	ErrorResponse
500	An unexpected error occurred.	ErrorResponse

5.2.4. Tags

- DataRepositoryService

5.3. Get a URL for fetching bytes.

```
GET /objects/{object_id}/access/{access_id}
```

5.3.1. Description

Returns a fully resolvable URL that can be used to fetch the actual object bytes.

5.3.2. Parameters

Type	Name	Description	Schema
Path	**access_id** *required*	An `access_id` from the `access_methods` list of a Data Object	string
Path	**object_id** *required*		string

5.3.3. Responses

HTTP Code	Description	Schema
200	The access URL was found successfully.	AccessURL
400	The request is malformed.	ErrorResponse
401	The request is unauthorized.	ErrorResponse
403	The requester is not authorized to perform this action.	ErrorResponse
404	The requested access URL wasn't found	ErrorResponse
500	An unexpected error occurred.	ErrorResponse

5.3.4. Tags

- DataRepositoryService

5.4. Get information about this implementation.

```
GET /service-info
```

5.4.1. Description

May return service version and other information.

5.4.2. Responses

HTTP Code	Description	Schema
200	Service information returned successfully	ServiceInfoResponse

5.4.3. Tags

- DataRepositoryService

Chapter 6. Definitions

6.1. AccessMethod

Name	Description	Schema
access_id *optional*	An arbitrary string to be passed to the /access path to get an `AccessURL`. This must be unique per object. Note that at least one of `access_url` and `access_id` must be provided.	string
access_url *optional*	An `AccessURL` that can be used to fetch the actual object bytes. Note that at least one of `access_url` and `access_id` must be provided.	AccessURL
region *optional*	Name of the region in the cloud service provider that the object belongs to. **Example** : `"us-east-1"`	string
type *required*	Type of the access method.	enum (s3, gs, ftp, sftp, http, https, nfs, globus, aspera, gsiftp, local)

6.2. AccessURL

Name	Description	Schema

headers
optional
	An optional list of headers to include in the HTTP request to `url`. These headers can be used to provide auth tokens required to fetch the object bytes.
	Example : `{`
	`"Authorization" : "Basic Z2E0Z2g6ZHJz"`
	`}`
		< string > array

url *required*	A fully resolvable URL that can be used to fetch the actual object bytes.	string

6.3. Bundle

Name	Description	Schema
aliases *optional*	A list of strings that can be used to identify this Data Bundle.	< string > array
checksums *required*	At least one checksum must be provided. The Data Bundle checksum is computed over all the checksums of the Data Objects that bundle contains.	< Checksum > array
created *required*	Timestamp of object creation in RFC3339.	string (date-time)
description *optional*	A human readable description.	string

Name	Description	Schema
id *required*	An identifier, unique to this Data Bundle	string
object_ids *required*	The list of Data Objects that this Data Bundle contains.	< string > array
system_metadata *optional*		SystemMetadata
updated *required*	Timestamp of update in RFC3339, identical to create timestamp in systems that do not support updates.	string (date-time)
user_metadata *optional*		UserMetadata
version *required*	A string representing a version, some systems may use checksum, a RFC3339 timestamp, or incrementing version number. For systems that do not support versioning please use your update timestamp as your version.	string

6.4. Checksum

Name	Description	Schema
checksum *required*	The hex-string encoded checksum for the Data.	string
type *optional*	The digest method used to create the checksum. If left unspecified md5 will be assumed. possible values: md5 # most blob stores provide a checksum using this multipart-md5 # multipart uploads provide a specialized tag in S3 sha256 sha512	string

6.5. ErrorResponse

An object that can optionally include information about the error.

Name	Description	Schema
msg *optional*	A detailed error message.	string
status_code *optional*	The integer representing the HTTP status code (e.g. 200, 404).	integer

6.6. GetBundleResponse

Name	Schema
bundle *optional*	Bundle

6.7. GetObjectResponse

Name	Schema
object *required*	Object

6.8. Object

Name	Description	Schema
access_methods *required*	The list of access methods that can be used to access the Data Object.	< AccessMethod > array
aliases *optional*	A list of strings that can be used to find this Data Object. These aliases can be used to represent the Data Object's location in a directory (e.g. "bucket/folder/file.name") to make Data Objects more discoverable. They might also be used to represent	< string > array
checksums *required*	The checksum of the Data Object. At least one checksum must be provided.	< Checksum > array
created *required*	Timestamp of object creation in RFC3339.	string (date-time)
description *optional*	A human readable description of the contents of the Data Object.	string
id *required*	An identifier unique to this Data Object.	string
mime_type *optional*	A string providing the mime-type of the Data Object. For example, "application/json".	string
name *optional*	A string that can be optionally used to name a Data Object.	string
size *required*	The computed size in bytes.	string (int64)
updated *optional*	Timestamp of update in RFC3339, identical to create timestamp in systems that do not support updates.	string (date-time)
version *optional*	A string representing a version.	string

6.9. ServiceInfoResponse

Placeholder for the Info Object

Name	Description	Schema
contact *optional*	Maintainer contact info	object
description *optional*	Service description	string
license *optional*	License information for the exposed API	object
title *optional*	Service name	string
version *required*	Service version	string

6.10. SystemMetadata

OPTIONAL
These values are reported by the underlying object store.
A set of key-value pairs that represent system metadata about the object.

Type : object

6.11. UserMetadata

OPTIONAL
A set of key-value pairs that represent metadata provided by the uploader.

Type : object

Chapter 7. Appendix: Motivation

Data sharing requires portable data, consistent with the FAIR data principles (findable, accessible, interoperable, reusable). Today’s researchers and clinicians are surrounded by potentially useful data, but often need bespoke tools and processes to work with each dataset. And today’s data publishers don’t have a reliable way to make their data useful to all (and only) the people they choose.

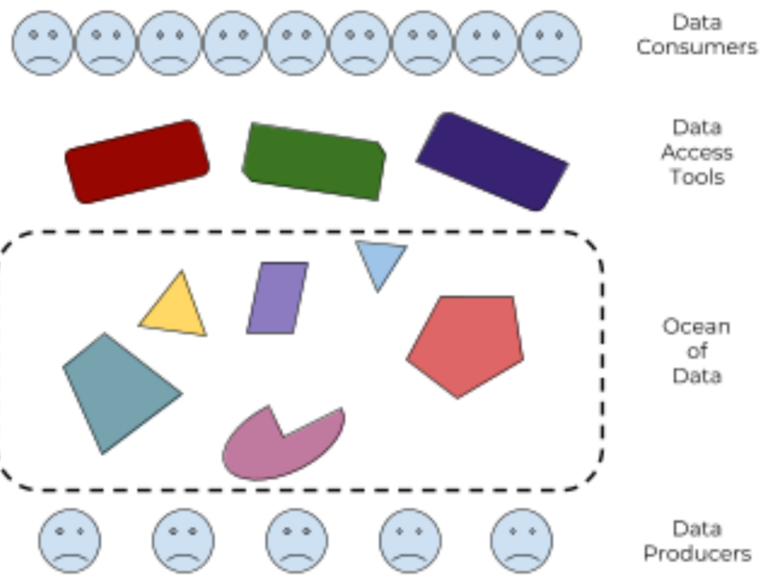

Figure 1: there’s an ocean of data, with many different tools to drink from it, but no guarantee that any tool will work with any subset of the data

We need a standard way for data producers to make their data available to data consumers, that supports the control needs of the former and the access needs of the latter. And we need it to be interoperable, so anyone who builds access tools and systems can be confident they’ll work with all the data out there, and anyone who publishes data can be confident it will work with all the tools out there.

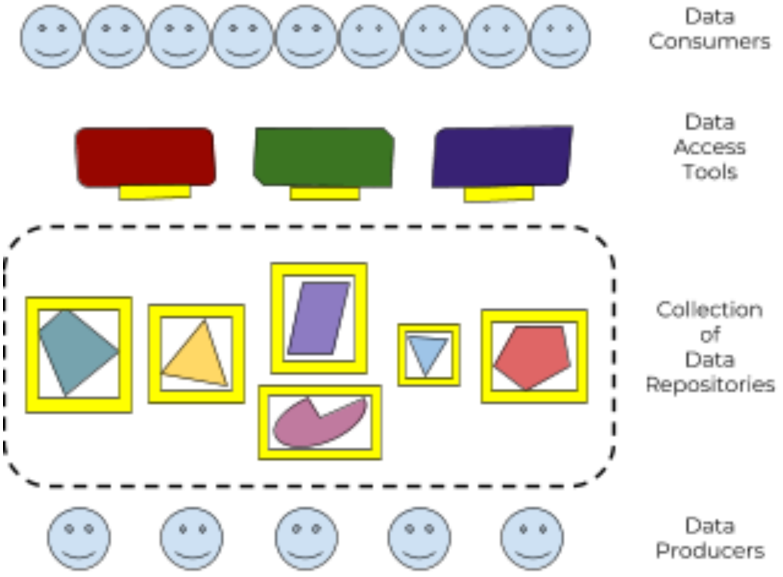

Figure 2: by defining a standard Data Repository API, and adapting tools to use it, every data publisher can now make their data useful to every data consumer

We envision a world where:

- there are many many **data consumers**, working in research and in care, who can use the tools of their choice to access any all data that they have permission to see
- there are many **data access tools** and platforms, supporting discovery, visualization, analysis, and collaboration
- there are many **data repositories**, each with their own policies and characteristics, which can be accessed by a variety of tools
- there are many **data publishing tools** and platforms, supporting a variety of data lifecycles and formats
- there are many many **data producers**, generating data of all types, who can use the tools of their choice to make their data as widely available as is appropriate

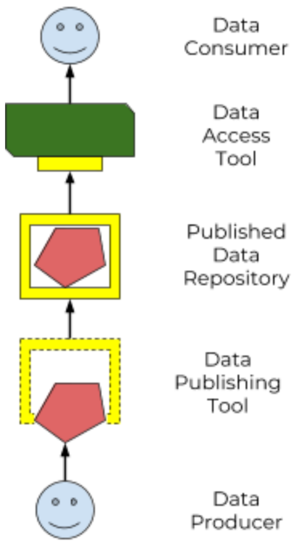

Figure 3: a standard Data Repository API enables an ecosystem of data producers and consumers

This spec defines a standard **Data Repository Service (DRS) API** (“the yellow box”), to enable that ecosystem of data producers and consumers. Our goal is that all data consumers need to know about a data repo is "here’s the DRS endpoint to access it", and all data publishers need to know about tapping into the world of consumption tools is "here’s how to tell it where my DRS endpoint lives".

7.1. Federation

The world’s biomedical data is controlled by groups with very different policies and restrictions on where their data lives and how it can be accessed. A primary purpose of DRS is to support unified access to disparate and distributed data. (As opposed to the alternative centralized model of "let’s just bring all the data into one single data repository”, which would be technically easier but is no more realistic than “let’s just bring all the websites into one single web host”.)

In a DRS-enabled world, tool builders don’t have to worry about where the data their tools operate on lives — they can count on DRS to give them access. And tool users only need to know which DRS server is managing the data they need, and whether they have permissions; they don’t have to worry about how to physically get access to, or (worse) make a copy of the data. For example, if I have appropriate permissions, I can run a pooled analysis where I run a single tool across data managed by different DRS servers, potentially in different locations.