Data Repository Service

Table of Contents

1. Overview . . . 1
   1.1. Version information . . . 1
   1.2. Contact information . . . 1
   1.3. License information . . . 1
   1.4. URI scheme . . . 1
   1.5. Tags . . . 1
   1.6. Consumes . . . 1
   1.7. Produces . . . 1
2. Introduction . . . 2
3. DRS API Principles . . . 3
   3.1. DRS IDs . . . 3
   3.2. DRS URIs . . . 3
   3.3. DRS Datatypes . . . 3
   3.4. Read-only . . . 3
   3.5. Standards . . . 3
4. Authorization & Authentication . . . 4
   4.1. Making DRS Requests . . . 4
   4.2. Fetching DRS Objects . . . 4
5. Paths . . . 5
   5.1. Get info about an `Object`. . . . 5
      5.1.1. Description . . . 5
      5.1.2. Parameters . . . 5
      5.1.3. Responses . . . 5
      5.1.4. Tags . . . 5
   5.2. Get a URL for fetching bytes. . . . 5
      5.2.1. Description . . . 5
      5.2.2. Parameters . . . 6
      5.2.3. Responses . . . 6
      5.2.4. Tags . . . 6
   5.3. Get information about this implementation. . . . 6
      5.3.1. Description . . . 6
      5.3.2. Responses . . . 6
      5.3.3. Tags . . . 6
6. Definitions . . . 7
   6.1. AccessMethod . . . 7
   6.2. AccessURL . . . 7
   6.3. Checksum . . . 7
   6.4. ContentsObject . . . 8

6.5. Error 8
6.6. Object 8
6.7. ServiceInfo 10
7. Appendix: Motivation 11
7.1. Federation 12

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 : https://www.ga4gh.org/terms-and-conditions/

1.4. URI scheme

BasePath : /ga4gh/drs/v1
Schemes : HTTPS

1.5. Tags

- DataRepositoryService

1.6. Consumes

- `application/json`

1.7. 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 responses. 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 v1 does NOT support semantics around multiple versions of an object. (For example, there's no notion of "get latest version" or "list all versions".) 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 URIs

For convenience, including when passing content references to a WES server, we define a URI syntax for DRS-accessible content. Strings of the form `drs://<server>/<id>` mean *"you can fetch the content with DRS id `<id>` from the DRS server at `<server>` ".*

For example, if a WES server was asked to process `drs://drs.example.org/314159`, it would know that it could issue a GET request to `http://drs.example.org/ga4gh/drs/v1/objects/314159` to learn how to fetch that object.

3.3. DRS Datatypes

DRS v1 supports two types of content:

- a *blob* is like a file — it's a single blob of bytes, represented by an `Object` without a `contents` array
- a *bundle* is like a folder — it's a collection of other DRS content (either blobs or bundles), represented by an `Object` with a `contents` array

3.4. 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.

3.5. Standards

The DRS API specification is written in OpenAPI and embodies a RESTful service philosophy. It uses JSON in requests and responses and standard 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. The `service-info` endpoint should provide sufficient information for a user to figure out how to authenticate with a DRS 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. Get info about an `Object`.

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

5.1.1. Description

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

5.1.2. Parameters

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

5.1.3. Responses

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

5.1.4. Tags

- DataRepositoryService

5.2. Get a URL for fetching bytes.

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

5.2.1. Description

Returns a URL that can be used to fetch the bytes of an `Object`.

This method only needs to be called when using an `AccessMethod` that contains an `access_id` (e.g., for servers that use signed URLs for fetching object bytes).

5.2.2. Parameters

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

5.2.3. Responses

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

5.2.4. Tags

- DataRepositoryService

5.3. Get information about this implementation.

```
GET /service-info
```

5.3.1. Description

May return service version and other information.

5.3.2. Responses

HTTP Code	Description	Schema
200	Service information returned successfully	ServiceInfo

5.3.3. Tags

- DataRepositoryService

Chapter 6. Definitions

6.1. AccessMethod

Name	Description	Schema
access_id *optional*	An arbitrary string to be passed to the `/access` method to get an `AccessURL`. This string 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, gsiftp, globus, htsget, https, file)

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. 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 etag # multipart uploads to blob stores sha256 sha512	string

6.4. ContentsObject

Name	Description	Schema
drs_uri *optional*	A list of full DRS identifier URI paths that may be used to obtain the object. These URIs may be external to this DRS instance. **Example** : `"drs://example.com/ga4gh/drs/v1/objects/{object_id}"`	< string > array
id *required*	A DRS identifier of an `Object` (either a single blob or a nested bundle).	string
name *required*	A name declared by the bundle author that must be used when materialising this object, overriding any name directly associated with the object itself. The name must be unique with the containing bundle. This string is made up of uppercase and lowercase letters, decimal digits, hypen, period, and underscore [A-Za-z0-9.-_]. See RFC 3986 § 2.3. All RFC 3986 § 3986 reserved characters including whitespace in the object name must be url-encoded [!*'();:@&=+$,/?#[]]. See RFC 3986 § 2.2.	string

6.5. Error

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. Object

Name	Description	Schema
access_methods *optional*	The list of access methods that can be used to fetch the `Object`. Required for single blobs; optional for bundles.	< AccessMethod > array
aliases *optional*	A list of strings that can be used to find other metadata about this `Object` from external metadata sources. These aliases can be used to represent secondary accession numbers or external GUIDs.	< string > array

Name	Description	Schema
checksums *required*	The checksum of the `Object`. At least one checksum must be provided. For blobs, the checksum is computed over the bytes in the blob. For bundles, the checksum is computed over a sorted concatenation of the checksums of its top-level contained objects (not recursive, names not included). The list of checksums is sorted alphabetically (hex-code) before concatenation and a further checksum is performed on the concatenated checksum value. For example, if a bundle contains blobs with the following checksums: md5(blob1) = 72794b6d md5(blob2) = 5e089d29 Then the checksum of the bundle is: md5(concat(sort(md5(blob1), md5(blob2)))) = md5(concat(sort(72794b6d, 5e089d29))) = md5(concat(5e089d29, 72794b6d)) = md5(5e089d2972794b6d) = f7a29a04	< Checksum > array
contents *optional*	If not set, this `Object` is a single blob. If set, this `Object` is a bundle containing the listed `ContentsObject` s (some of which may be further nested).	< ContentsObject > array
created *required*	Timestamp of object creation in RFC3339.	string (date-time)
description *optional*	A human readable description of the `Object`.	string
id *required*	An identifier unique to this `Object`.	string
mime_type *optional*	A string providing the mime-type of the `Object`. **Example** : `"application/json"`	string
name *optional*	A string that can be used to name an `Object`. This string is made up of uppercase and lowercase letters, decimal digits, hypen, period, and underscore [A-Za-z0-9.-_]. See RFC 3986 § 2.3. All RFC 3986 § 3986 reserved characters including whitespace in the object name must be url-encoded [!*'();:@&=+$,/?#[]]. See RFC 3986 § 2.2.	string
size *required*	For blobs, the blob size in bytes. For bundles, the cumulative size, in bytes, of items in the `contents` field.	integer (int64)
updated *optional*	Timestamp of `Object` update in RFC3339, identical to create timestamp in systems that do not support updates.	string (date-time)

Name	Description	Schema
version *optional*	A string representing a version. (Some systems may use checksum, a RFC3339 timestamp, or an incrementing version number.)	string

6.7. ServiceInfo

Useful information about the running service.

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

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. Today's data publishers don't have a reliable way to make their data useful to all (and only) the people they choose. And today's data controllers are tasked with implementing standard controls of non-standard mechanisms for data access.

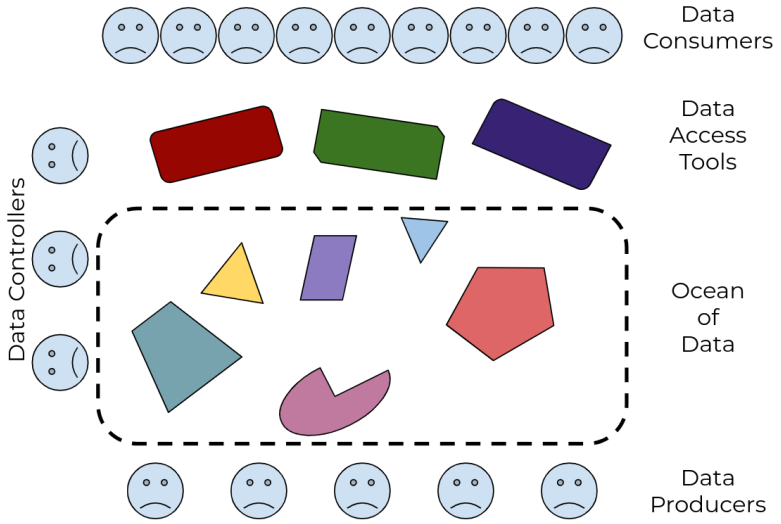

Figure 1: there's an ocean of data, with many different tools to drink from it, but no guarantee that any tool will work with any subset of the data

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.

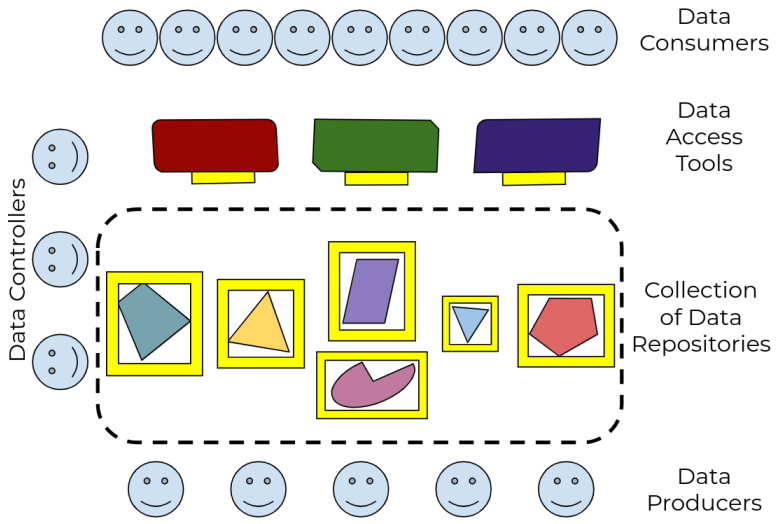

Figure 2: by defining a standard Data Repository API, and adapting tools to use it, every data publisher can now make their data useful to every data consumer

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 and 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

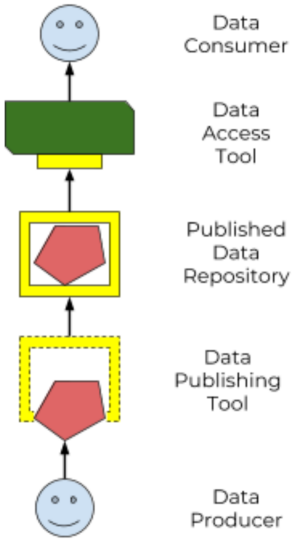

Figure 3: a standard Data Repository API enables an ecosystem of data producers and consumers

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 the only thing data consumers need to know about a data repo is *"here’s the DRS endpoint to access it"*, and the only thing data publishers need to know to tap 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 permission to access it; 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.