Data Repository Service

Table of Contents

Chapter 1. Overview

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

1.1. Version information

Version : 1.0.0

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 objects in a single, standard way regardless of where they are stored and how they are managed. 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, how they can be pointed to using URIs, and how clients can use these URIs to ultimately make successful DRS API requests. This document also describes the DRS API in detail and provides information on the specific endpoints, request formats, and responses. This specification is intended for developers of DRS-compatible services and of clients that will call these DRS services.

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 strings made up of uppercase and lowercase letters, decimal digits, hypen, period, underscore and tilde [A-Za-z0-9.-_~]. See RFC 3986 § 2.3.
- DRS IDs can contain other characters, but they MUST be encoded into valid DRS IDs whenever exposed by the API. This is because non-encoded IDs may interfere with the interpretation of the `objects/{id}/access` endpoint. To overcome this limitation use percent-encoding of the ID, see RFC 3986 § 2.4
- 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 implementations MAY have more than one ID that maps to the same object.
- DRS version 1.x 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.

3.2. DRS URIs

For convenience, including when passing content references to a WES server, we define a URI scheme for DRS-accessible content. See RFC 3986 § 3.1. *We feel it is important to introduce a DRS scheme for URIs since it signals to systems consuming these URIs that the response they will ultimately receive, once transforming the URI to a fetchable URL, will be a DRS JSON.*

There are two styles of DRS URIs: Compact Identifier-based and Hostname-based, both use the `drs://` URI scheme. Different systems may choose to share one or both styles of DRS URIs but DRS clients must support resolving both types.

3.2.1. Compact Identifier-based DRS URIs

Compact identifiers refer to locally-unique persistent identifiers that have been namespaced to provide global uniqueness. See "Uniform resolution of compact identifiers for biomedical data" for an excellent introduction to this topic. We support the use of compact identifiers in DRS URIs since many resources in the research community issue these identifiers in a variety of formats. We leverage the resolver registry services of identifiers.org or n2t.net (Name-To-Thing) to support any registered form of compact identifiers (Arks, DOIs, Data GUIDs, etc), allowing for the resolution of DRS objects without needing to use a hostname in the URI.

> **Note: DRS Service Implementations Prefix Registration**
>
> If your DRS implementation will issue IDs based on compact identifiers, and will pass around DRS URIs using these compact identifiers, you **must** register your prefix (which consists of `[provider_code/]namespace`) on identifiers.org/n2t.net. If you don't, DRS clients will not know how to resolve your compact identifiers and, ultimately, generate a valid DRS URL that clients can access.

Translating the identifiers.org/n2t.net CURIE format to a DRS compact identifier-based URI we get the following form. Together, provider code and the namespace are referred to as the *prefix*. The provider code is optional and is used by identifiers.org/n2t.net for compact identifier resolver mirrors:

```
drs://[provider_code/]namespace:accession
```

TIP Much more information is provided in the Appendix: Compact Identifier-Based URIs including details on the structure of compact identifiers, how to register prefixes, how clients resolve compact identifier-based DRS URIs, possible caching strategies, and security considerations.

3.2.2. Hostname-based DRS URIs

Hostname-based DRS URIs are simpler than compact identifier-based URIs. They contain the DRS server name and the DRS ID only and can be converted directly into a fetchable URL based on a simple rule. They take the form:

```
drs://<hostname>/<id>
```

Hostname-based DRS URIs are less resistant to future project/domain name changes than compact identifiers. But they do provide a more direct way of pointing to a DRS object which can have benefits. The fact that they can be resolved using a simple rule means a DRS client can skip the extra overhead of a prefix lookup as is done for compact identifier-based URIs. This can translate to possibly greater performance and/or security since the DRS server hostname is explicitly specified and this avoids the lookup of a resolver through a third-party service (identifiers.org/n2t.net).

TIP Much more information is provided in the Appendix: Hostname-Based URIs including details on how to resolve hostname-based DRS URIs. It also includes information on how hostname-based DRS URI resolution to URLs will work in the future, when the DRS v2 major release happens.

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 a `DrsObject` without a `contents` array
- a *bundle* is like a folder — it's a collection of other DRS content (either blobs or bundles), represented by a `DrsObject` 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 on port 443 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. GA4GH recommends that DRS implementations use an OAuth 2.0 bearer token, although they can choose other mechanisms if appropriate.

4.2. Fetching DRS Objects

The DRS API allows implementers to support a variety of different content access policies, depending on what `AccessMethod` records 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)

DRS implementers should ensure their solutions restrict access to targets as much as possible, detect attempts to exploit through log monitoring, and they are prepared to take action if an exploit in their DRS implementation is detected.

Chapter 5. Paths

5.1. Get info about a `DrsObject`.

```
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	Description	Schema	Default
Path	**object_id** *required*		string	
Query	**expand** *optional*	If false and the object_id refers to a bundle, then the ContentsObject array contains only those objects directly contained in the bundle. That is, if the bundle contains other bundles, those other bundles are not recursively included in the result. If true and the object_id refers to a bundle, then the entire set of objects in the bundle is expanded. That is, if the bundle contains aother bundles, then those other bundles are recursively expanded and included in the result. Recursion continues through the entire sub-tree of the bundle. If the object_id refers to a blob, then the query parameter is ignored.	boolean	`"false"`

5.1.3. Responses

HTTP Code	Description	Schema
200	The `DrsObject` was found successfully.	DrsObject

HTTP Code	Description	Schema
202	The operation is delayed and will continue asynchronously. The client should retry this same request after the delay specified by Retry-After header. **Headers** : `Retry-After` (integer (int64)) : Delay in seconds. The client should retry this same request after waiting for this duration. To simplify client response processing, this must be an integral relative time in seconds. This value SHOULD represent the minimum duration the client should wait before attempting the operation again with a reasonable expectation of success. When it is not feasible for the server to determine the actual expected delay, the server may return a brief, fixed value instead.	No Content
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 `DrsObject` 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 a `DrsObject`.

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 a `DrsObject`	string
Path	**object_id** *required*	An `id` of a `DrsObject`	string

5.2.3. Responses

HTTP Code	Description	Schema
200	The access URL was found successfully.	AccessURL
202	The operation is delayed and will continue asynchronously. The client should retry this same request after the delay specified by Retry-After header. **Headers** : `Retry-After` (integer (int64)) : Delay in seconds. The client should retry this same request after waiting for this duration. To simplify client response processing, this must be an integral relative time in seconds. This value SHOULD represent the minimum duration the client should wait before attempting the operation again with a reasonable expectation of success. When it is not feasible for the server to determine the actual expected delay, the server may return a brief, fixed value instead.	No Content
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

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 within the scope of a single 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

Name	Description	Schema
type *required*	The digest method used to create the checksum. The value (e.g. `sha-256`) SHOULD be listed as `Hash Name String` in the IANA Named Information Hash Algorithm Registry. Other values MAY be used, as long as implementors are aware of the issues discussed in RFC6920. GA4GH may provide more explicit guidance for use of non-IANA-registered algorithms in the future. Until then, if implementors do choose such an algorithm (e.g. because it's implemented by their storage provider), they SHOULD use an existing standard `type` value such as `md5`, `etag`, `crc32c`, `trunc512`, or `sha1`. **Example** : `"sha-256"`	string

6.4. ContentsObject

Name	Description	Schema
contents *optional*	If this ContentsObject describes a nested bundle and the caller specified "?expand=true" on the request, then this contents array must be present and describe the objects within the nested bundle.	< ContentsObject > array
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://drs.example.org/314159"`	< string > array
id *optional*	A DRS identifier of a `DrsObject` (either a single blob or a nested bundle). If this ContentsObject is an object within a nested bundle, then the id is optional. Otherwise, the id is required.	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 portable filenames.	string

6.5. DrsObject

Name	Description	Schema
access_methods *optional*	The list of access methods that can be used to fetch the `DrsObject`. Required for single blobs; optional for bundles.	< AccessMethod > array

Name	Description	Schema
aliases *optional*	A list of strings that can be used to find other metadata about this `DrsObject` from external metadata sources. These aliases can be used to represent secondary accession numbers or external GUIDs.	< string > array
checksums *required*	The checksum of the `DrsObject`. 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 `DrsObject` is a single blob. If set, this `DrsObject` is a bundle containing the listed `ContentsObject` s (some of which may be further nested).	< ContentsObject > array
created_time *required*	Timestamp of content creation in RFC3339. (This is the creation time of the underlying content, not of the JSON object.)	string (date-time)
description *optional*	A human readable description of the `DrsObject`.	string
id *required*	An identifier unique to this `DrsObject`.	string
mime_type *optional*	A string providing the mime-type of the `DrsObject`. **Example** : `"application/json"`	string
name *optional*	A string that can be used to name a `DrsObject`. This string is made up of uppercase and lowercase letters, decimal digits, hypen, period, and underscore [A-Za-z0-9.-_]. See portable filenames.	string

Name	Description	Schema
self_uri *required*	A drs:// URI, as defined in the DRS documentation, that tells clients how to access this object. The intent of this field is to make DRS objects self-contained, and therefore easier for clients to store and pass around. **Example** : `"drs://drs.example.org/314159"`	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_time *optional*	Timestamp of content update in RFC3339, identical to `created_time` in systems that do not support updates. (This is the update time of the underlying content, not of the JSON object.)	string (date-time)
version *optional*	A string representing a version. (Some systems may use checksum, a RFC3339 timestamp, or an incrementing version number.)	string

6.6. 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

Chapter 7. Appendix: Compact Identifier-Based URIs

Compact identifiers refer to locally-unique persistent identifiers that have been namespaced to provide global uniqueness. See "Uniform resolution of compact identifiers for biomedical data" for an excellent introduction to this topic. By using compact identifiers in DRS URIs, along with a resolver registry (identifiers.org/n2t.net), systems can identify the current resolver when they need to translate a DRS URI into a fetchable URL. This allows a project to issue compact identifiers in DRS URIs and not be concerned if the project name or DRS hostname changes in the future, the current resolver can always be found through the identifiers.org/n2t.net registries. Together the identifiers.org/n2t.net systems support the resolver lookup for over 700 compact identifiers formats used in the research community, making it possible for a DRS server to use any of these as DRS IDs (or to register a new compact identifier type and resolver service of their own).

We use a DRS URI scheme rather than Compact URIs (CURIEs) directly since we feel that systems consuming DRS objects will be able to better differentiate a DRS URI. CURIEs are widely used in the research community and we felt the fact that they can point to a wide variety of entities (HTML documents, PDFs, identities in data models, etc) makes it more difficult for systems to unambiguously identify entities as DRS objects.

Still, to make compact identifiers work in DRS URIs we leverage the CURIE format used by identifiers.org/n2t.net. Compact identifiers have the form:

```
prefix:accession
```

The prefix can be divided into an optional `provider_code` and `namespace`. The `accession` here is an Ark, DOI, Data GUID, or another issuers's local ID for the object being pointed to:

```
[provider_code/]namespace:accession
```

Both the `provider_code` and `namespace` disallow spaces or punctuation, only lowercase alphanumerical characters, underscores and dots are allowed.

Examples include (from n2t.net):

```
PDB:2gc4
Taxon:9606
DOI:10.5281/ZENODO.1289856
ark:/47881/m6g15z54
IGSN:SSH000SUA
```

TIP DRS URIs using compact identifiers with resolvers registered in identifiers.org/n2t.net can be distinguished from the hostname-based DRS URIs below based on the required ":" which is not allowed in hostname-based URI.

TIP The CURIE format used by identifiers.org/n2t.net does not percent-encode reserved URI characters but, instead, relies on the first ":" character to separate prefix from accession. Since these accessions can contain any characters, and characters like "/" will interfere with DRS API calls, you *must* percent encode the accessions extracted from DRS compact identifier-based URIs when using as DRS IDs in DRS GET requests. For more information see the Note "DRS Client Compact Identifier-Based URI Resolution Process".

See the documentation on n2t.net and identifiers.org for much more information on the compact identifiers used there and details about the resolution process.

Note: DRS Client Compact Identifier-Based URI Resolution Process

A DRS client identifies the a DRS URI compact identifier components using the first occurance of "/" (optional) and ":" characters. These are not allowed inside the provider_code (optional) or the namespace. The ":" character is not allowed in a Hostname-based DRS URI, providing a convenient mechanism to differentiate them. Once the provider_code (optional) and namespace are extracted from a DRS compact identifier-based URI, a client can use services on identifiers.org to identify available resolvers.

Let's look at a specific example DRS compact identifier-based URI that uses DOIs, a popular compact identifier, and walk through the process that a client would use to resolve it. Starting with the DRS URI:

```
drs://doi:10.5072/FK2805660V
```

with a namespace of "doi", the following GET request will return information about the namespace:

```
GET
https://registry.api.identifiers.org/restApi/namespaces/search/findByPrefix?prefi
x=doi
```

This information then points to resolvers for the "doi" namespace (this "doi" namespace was assigned a namespace ID of 75 by identifiers.org, this "id" has nothing to do with compact identifier accessions or DRS IDs):

GET https://registry.api.identifiers.org/restApi/resources/search/findAllByNamespaceId?id=75

This returns enough information to, ultimately, identify one or more resolvers and each have a URL pattern that, for DRS-supporting systems, provides a URL template for making a successful DRS GET request. For example, the DOI urlPattern is:

```
urlPattern: "https://doi.org/{$id}"
```

If applicable, a provide code can be supplied in the above requests to specify a particular mirror if there are multiple resolvers for this namespace. In the case of DOIs you only get a single resolver.

Given this information you now know you can make a GET on the URL:

```
GET https://doi.org/10.5072/FK2805660V
```

The URL above is valid for a DOI object but it is not actually a DRS server! Instead, it redirects to a DRS server through a series of HTTP redirects. This is likely to be common when working with existing compact identifiers like DOIs or Arks. Regardless, the redirect should eventually *lead to a DRS URL that percent-encodes the access as a DRS ID in a DRS object API call. For a hypothetical example, here's what a redirect to a DRS API might look like for the Data GUIDs compact identifiers who host a DRS server.*

```
GET https://dataguids.org/ga4gh/drs/v1/objects/dg.4503%2F00e6cfa9-a183-42f6-bb44-b70347106bbe
```

IDs in DRS hostname-based URIs/URLs are always percent-encoded to eliminate ambiguity even though the DRS compact identifier-based URIs and the identifier.orgs API do not percent-encode accessions. This was done in order to 1) follow the CURIE conventions of identifiers.org/n2t.net for compact identifier-based DRS URIs and 2) to aid in readability for users who understand they are working with compact identifiers. **The general rule of thumb, when using a compact identifier accession as a DRS ID in a DRS API call, make sure to percent-encode it. An easy way to handle this is to get the initial DRS object JSON response from whatever redirects the compact identifier resolves to, then look for the `self_uri` in the JSON which will give you the correctly percent-encoded DRS ID for subsequent DRS API calls such as the access method.**

Note: Registering a new Compact Identifier

See the documentation on n2t.net and identifiers.org for. You can register new prefixes (or mirrors by adding resource provider codes) for free using a simple online form. Keep in mind, while anyone can register prefixes, the identifiers.org/n2t.net sites do basic hand curation to verify new prefix and resource (provider code) requests. See those sites for more details on their security practices.

Note: Caching and Security with Compact Identifiers

As mentioned earlier identifiers.org/n2t.net resolver records do not change frequently. This approach is useful for caching resolver records and their URL patterns for performance reasons. Builders of systems that use compact identifier-based DRS URIs should cache prefix resolver records from identifiers.org/n2t.net and occasionally refresh the records (such as every 24 hours). The implementation of this approach is up to the system builder but it will ultimately reduce the burden on identifiers.org/n2t.net. System builders may also choose to cache the compact identifier accession to final DRS URL records as well, but that decision/design is at the discretion of the system builder.

As mentioned earlier identifiers.org/n2t.net performs some basic verification of new prefixes and provider code mirror registrations on their sites. However, system builders may have certain compliance requirements and regulations that prohibit relying on an external site for resolving compact identifiers. In this case, systems under these security and compliance constraints may which to either whitelist certain compact identifier resolvers and/or vet records from identifiers.org/n2t.net before enabling in their system.

Chapter 8. Appendix: Hostname-Based URIs

Note: DRS Client Hostname-Based URI Resolution Process

Strings of the form `drs://<hostname>/<id>` mean *“you can fetch the content with DRS id `<id>` from the DRS server at `<hostname>` ”*. 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:

```
https://drs.example.org/ga4gh/drs/v1/objects/314159
```

to learn how to fetch that data object via one of the available access approaches.

The protocol is always https and the port is always the standard 443 SSL port. It would be invalid to include, for example, a different port in the DRS hostname-based URI.

In hostname-based DRS URIs, the ID is always percent-encoded to ensure special characters do not interfere with subsequent DRS endpoint calls. As such, ":" is not allowed in the URI and is a convenient way of differentiating from a compact identifier-based DRS URI. Also, if a given DRS service implementation uses compact identifier accessions as their DRS IDs, they must be percent encoded before using them as DRS IDs in hostname-based DRS URIs.

Note: Service Registry/Info and Future Versions of DRS

In the future, as newer major versions of DRS are released, the ability to look at a hostname-based DRS URI and derive a valid GET URL will not be possible. Multiple versions of DRS on different URL paths may be supported on the same server. We expect to add support for service-registry and service-info endpoints in future releases of DRS. Using the hostname in the DRS URI, plus information in the service-registry standard endpoint (which lead to service-info endpoints on that server) a client will be able to discover enough information to translate a DRS hostname-based URI into a valid URL. For, now we assume a rules-based translation to the pattern <code><a href="https://<hostname>/ga4gh/drs/v1/objects/<id&gt" class="bare">https://<hostname>/ga4gh/drs/v1/objects/<id&gt;</code>

Chapter 9. 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.

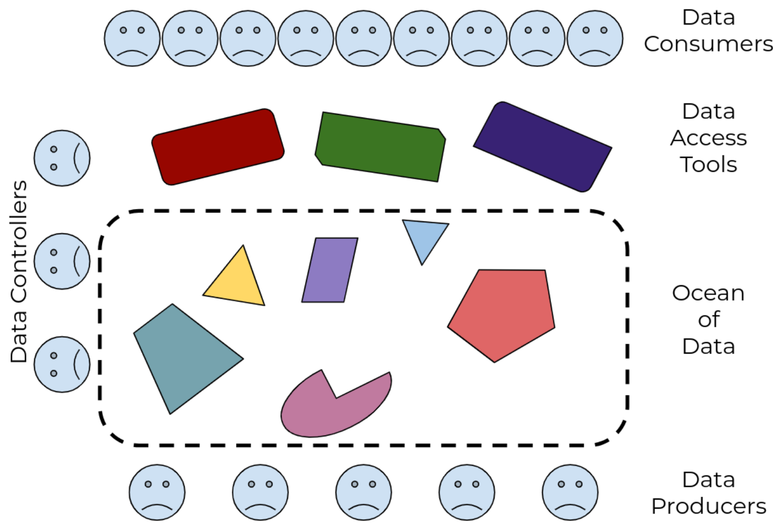

Figure 1: there's an ocean of data, with many different tools to drink from it, but no guarantee that any tool will work with any subset of the data

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.

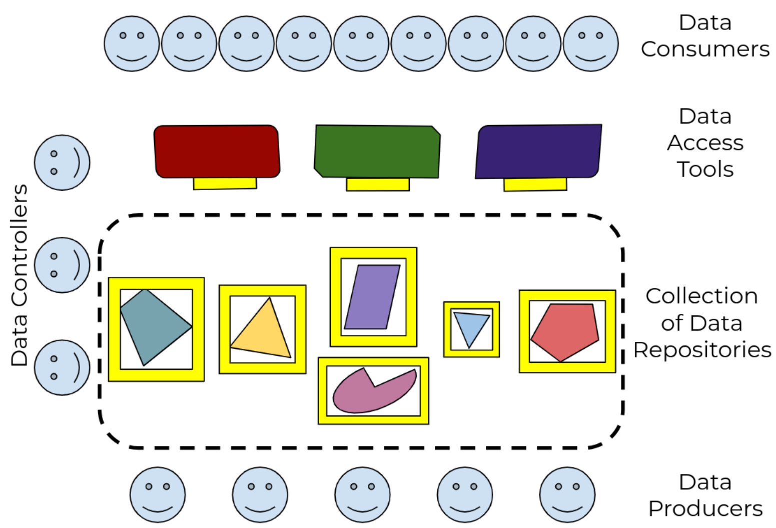

Figure 2: by defining a standard Data Repository API, and adapting tools to use it, every data publisher can now make their data useful to every data consumer

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

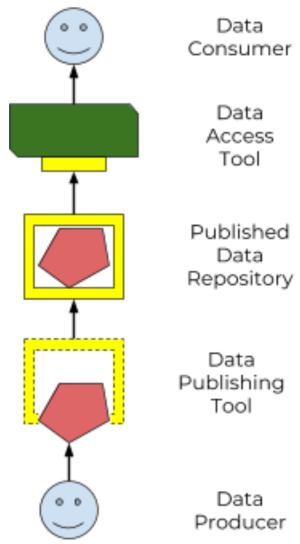

Figure 3: a standard Data Repository API enables an ecosystem of data producers and consumers

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"*.

9.1. 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.