# 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. This document describes the DRS API and provides details on the specific endpoints, request formats, and responses. We also describe a convention for pointing to DRS data objects using a custom URI scheme and how clients can use this to ultimately make successful DRS API requests. This specification 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, how they can be pointed to using URIs, 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 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-URL 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 felt it was 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.* If we had gone with a Compact URI (CURIE) we felt that this would have been more difficult for systems consuming DRS objects to understand and differentiate them given the ubiquitous use of CURIEs in the research community and the fact that CURIEs can point to a wide variety of entities (HTML documents, PDFs, identities in data models, etc).

There are two styles of DRS URIs: Compact Identifier-based and Hostname-based, both use the `drs://` URI scheme.

### 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.

When combined with the resolver registry services of identifiers.org or n2t.net (Name-To-Thing), we can support any registered form of compact identifiers (Arks, DOIs, Data GUIDs, etc) and also allow for the resolution of DRS objects without needing to use a hostname in the URI. By using compact identifiers with a resolver registry, systems using DRS URIs can identify the current resolver when needed. This allows a project to issue compact identifiers in DRS URIs and not be

concerned if the project name or DRS hostname change in the future since 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.

To make this work 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 the 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
```

### Note: DRS Service Implementers Prefix Registration

If your DRS implementation will issue IDs based on compact identifiers, and pass around DRS URIs using these compact identifiers, you **must** register your prefix 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 (provider code is optional):

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

| | |
|---|---|
| **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 them in DRS GET requests. For more information see the Note "DRS Client Compact Identifier Resolution Process" below.

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. You can also register new prefixes (or mirrors by adding resource providers) 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: 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 in provider_code (optional) or the namespace. The ":" character is also 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 URI, a client can use services on identifiers.org to identify available resolvers. For example, for a Data GUID namespace of "dg" the following GET request will return information about the namespace:

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

This information then points to resolvers for the "dg" namespace (assuming the namespace ID identified by the response to the above GET request for Data GUIDs is 1234):

```
GET
https://registry.api.identifiers.org/restApi/resources/search/findAllByNamespaceI
d?id=1234
```

This returns enough information to, ultimately, identify one or more resolvers and each have a URL prefix that, for DRS-supporting systems, tells how to make a successful DRS GET request.

Walking through a hypothetical Data GUID example compact identifier DRS URI's resolution:

```
drs://dg:4503/00e6cfa9-a183-42f6-bb44-b70347106bbe
```

Looking up the resolver on identifiers.org would tell you the URL pattern to get a DRS response would be:

```
https://dataguids.org/ga4gh/drs/v1/objects/dg.{$id}
```

And that can then be translated to the following using the accession parsed from the compact identifier that has been percent-encoded since DRS IDs must only use non-reserved URI characters:

```
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 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.*

**Please keep in mind identifiers.org/n2t.net does not support directly resolving percent-encoded accessions. So we recommend this approach above for DRS clients to looking up resolvers on these registries and then make a valid DRS GET request directly, using the percent-encoded accession as the DRS ID in the GET request. This approach is also useful for caching resolvers and their URL patterns for performance reasons since this information is unlikely to change frequently.**

### 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 convention. They take the form:

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

#### 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 IDs in hostname-based DRS URIs. For the earlier data GUID compact identifier DRS URI example, the hostname URI equivalent might look like: `drs://dataguids.org/dg.4503%2F00e6cfa9-a183-42f6-bb44-b70347106bbe`. Notice, when treated as a DRS ID, the compact identifier accession is URI percent encoded.

Hostname-based DRS URIs are less resistant to future project/domain name changes than compact identifiers. But they do provide a more explicit 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 DRS server lookup as is done for compact identifier-based URIs. This can translate to greater performance and also, possibly, security since it avoids the lookup of a resolver through a separate service (identifiers.org/n2t.net).

**Note: Service Registry/Info and Future Versions of DRS**

In the future, as newer 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 <a href="https://github.com/ga4gh-discovery/ga4gh-service-registry">service-registry</a> and <a href="https://github.com/ga4gh-discovery/ga4gh-service-info">service-info</a> 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 <code><a href="https://<hostname>/ga4gh/drs/v1/objects/<id>" class="bare">https://<hostname>/ga4gh/drs/v1/objects/<id&gt</a>;</code>

## 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` 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)

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**<br>*required* | | string | |
| **Query** | **expand**<br>*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.<br>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.<br>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.<br>**Headers** :<br>`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**<br>*required* | An `access_id` from the `access_methods` list of a `DrsObject` | string |
| **Path** | **object_id**<br>*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.<br>**Headers** :<br>`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**<br>*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**<br>*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**<br>*optional* | Name of the region in the cloud service provider that the object belongs to.<br>**Example** : `"us-east-1"` | string |
| **type**<br>*required* | Type of the access method. | enum (s3, gs, ftp, gsiftp, globus, htsget, https, file) |

## 6.2. AccessURL

| Name | Description | Schema |
|---|---|---|
| **headers**<br>*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.<br>**Example** : `{`<br>`"Authorization" : "Basic Z2E0Z2g6ZHJz"`<br>`}` | < string > array |
| **url**<br>*required* | A fully resolvable URL that can be used to fetch the actual object bytes. | string |

## 6.3. Checksum

| Name | Description | Schema |
|---|---|---|
| **checksum**<br>*required* | The hex-string encoded checksum for the data | string |

| Name | Description | Schema |
| --- | --- | --- |
| **type**<br>*required* | The digest method used to create the checksum.<br><br>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.<br><br>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`.<br>**Example** : `"sha-256"` | string |

## 6.4. ContentsObject

| Name | Description | Schema |
| --- | --- | --- |
| **contents**<br>*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**<br>*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.<br>**Example** : `"drs://drs.example.org/314159"` | < string > array |
| **id**<br>*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**<br>*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**<br>*optional* | The list of access methods that can be used to fetch the `DrsObject`.<br>Required for single blobs; optional for bundles. | < AccessMethod > array |

| Name | Description | Schema |
| --- | --- | --- |
| **aliases**<br>*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**<br>*required* | The checksum of the `DrsObject`. At least one checksum must be provided.<br>For blobs, the checksum is computed over the bytes in the blob.<br><br>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.<br><br>For example, if a bundle contains blobs with the following checksums:<br>md5(blob1) = 72794b6d<br>md5(blob2) = 5e089d29<br><br>Then the checksum of the bundle is:<br>md5( concat( sort( md5(blob1), md5(blob2) ) ) )<br>= md5( concat( sort( 72794b6d, 5e089d29 ) ) )<br>= md5( concat( 5e089d29, 72794b6d ) )<br>= md5( 5e089d2972794b6d )<br>= f7a29a04 | < Checksum > array |
| **contents**<br>*optional* | If not set, this `DrsObject` is a single blob.<br>If set, this `DrsObject` is a bundle containing the listed `ContentsObject` s (some of which may be further nested). | < ContentsObject > array |
| **created_time**<br>*required* | Timestamp of content creation in RFC3339.<br>(This is the creation time of the underlying content, not of the JSON object.) | string (date-time) |
| **description**<br>*optional* | A human readable description of the `DrsObject`. | string |
| **id**<br>*required* | An identifier unique to this `DrsObject`. | string |
| **mime_type**<br>*optional* | A string providing the mime-type of the `DrsObject`.<br>**Example** : `"application/json"` | string |
| **name**<br>*optional* | A string that can be used to name a `DrsObject`.<br>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**<br>*required* | A drs:// URI, as defined in the DRS documentation, that tells clients how to access this object.<br>The intent of this field is to make DRS objects self-contained, and therefore easier for clients to store and pass around.<br>**Example** : `"drs://drs.example.org/314159"` | string |
| **size**<br>*required* | For blobs, the blob size in bytes.<br>For bundles, the cumulative size, in bytes, of items in the `contents` field. | integer (int64) |
| **updated_time**<br>*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**<br>*optional* | A string representing a version.<br>(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**<br>*optional* | A detailed error message. | string |
| **status_code**<br>*optional* | The integer representing the HTTP status code (e.g. 200, 404). | integer |

# 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.

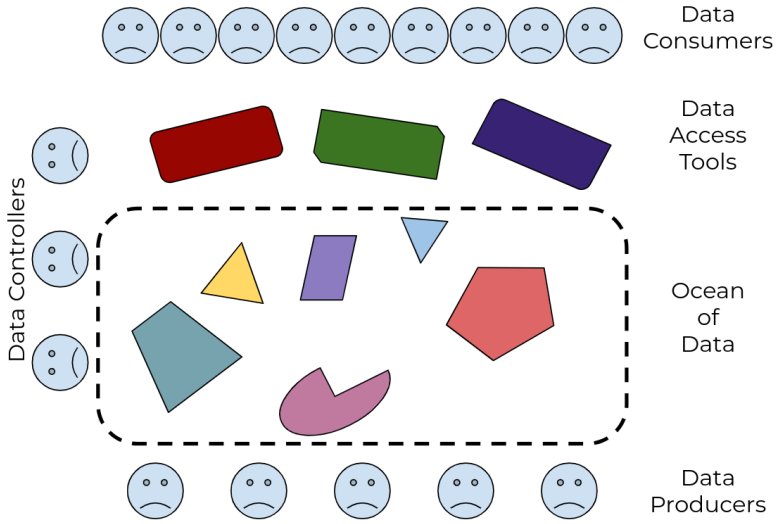


*Figure 1: there's an ocean of data, with many different tools to drink from it, but no guarantee that any tool will work with any subset of the data*

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.

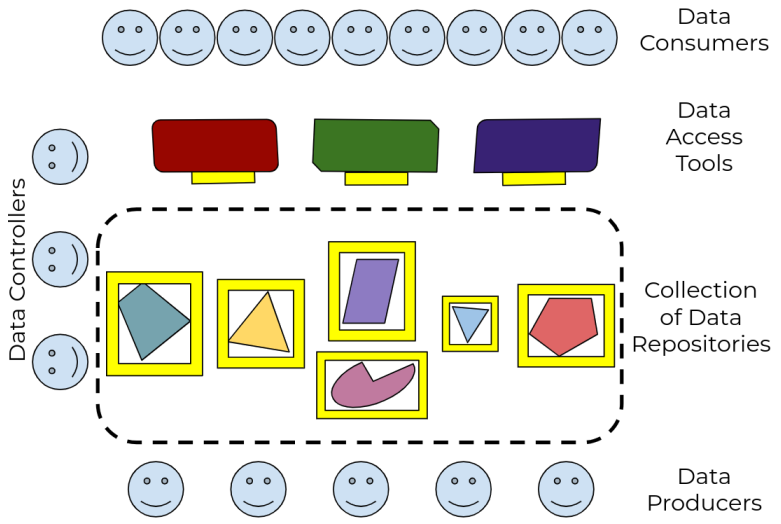


*Figure 2: by defining a standard Data Repository API, and adapting tools to use it, every data publisher can now make their data useful to every data consumer*

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

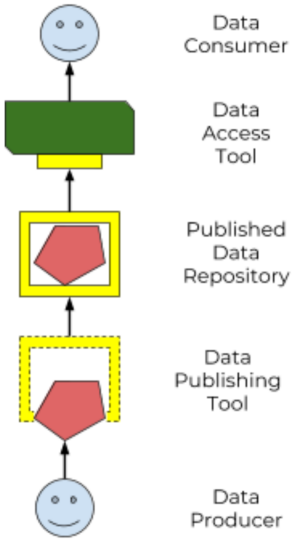


*Figure 3: a standard Data Repository API enables an ecosystem of data producers and consumers*

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.