Data management with ArtifactDB

Technical Design & Usage

Sébastien Lelong / DSSC

2022-12-14

Contents

Acknowledgments

First things first, let's go over the team and other contributors behind these backend systems:

TODO

Introduction

ArtifactDB is an "umbrella" name describing a type of API, built on top of the same open sourced framework. The concept is simple: secure storage of arbitrary data along with searchable metadata. ArtifactDB instances live close to the business and help collecting, organizing, cataloging rich domain-specific metadata. The data itself is by design treated as a "blob"

All ArtifactDB instances share the same features:

- Metadata must follow pre-defined JSON schemas. These schemas correspond to data types. They are converted into "models" used to make this metadata searchable through an efficient indexing engine (Elasticsearch)
- Authentication is based on JWT tokens, traditionally provided by an OpenID provider (based on the standard OAuth2.0)
- Fine-grained permissions can be defined using a Role Base Access Control (RBAC) pattern, based on unixID, distribution lists, or AD groups.
- Data and metadata are organized and grouped as projects, with versioning support (optional automatic provisioning of project identifiers and versions)
- Events are published during the data lifecycle, allowing users and other systems to be notified and to react as needed.
- Each API provides unique Genomics Platform Resource Names, or GPRNs to easily refer to any given resources within the GP (artifacts, projects, versions, changelog, documentation, etc…)
- Extensible with backend plugins, which can periodically run based on a schedule or based on certain events happening internally within ArtifactDB instances.
- Deployed as high performance, responsive and scalable REST APIs, built on top of Kubernetes, in the cloud.

Usage

TODO: REST API and admin shell

Upload artifacts

This section describes how to upload artifacts, ie. data and metadata files, to and ArtifactDB API. This is a multi-step process, during which the project enters different states.

Generally speaking, in a traditional setup, uploading artifacts involves requesting the instance to provision a project identifiers, or a new version within an existing project. This implies the instance itself exhibits a feature called `auto-provisioning`, handled by a component named the sequence manager, where provisioning and versioning is under the control of the instance itself. There are other use cases though where this "source" of IDs and versions is external to the instance. In that case, the sequence component is not used but instead, on the client side, the provisioning is performed *before* starting the upload process.

External and auto-provisioning

There are multiple REST endpoints involved in the uploading process, depending on whether the instance is using auto- or external IDs and version provisioning.

- `POST /projects/upload` is used to ask the instance to provision both a project ID, as well as a first version.
- `POST /projects/{project_id}/upload` is used to provision a new version within an existing project. It will fail if that project doesn't exist.
- `POST /projects/{project_id}/version/{version}/upload`, see below the different use cases

The first two endpoints require, by default, the role `creator`, to limit, if necessary, the process of project creation to a set of users. Upon project and/or version provisioning, the endpoint redirects to the same endpoint, the third one, `POST /projects/{project_id}/version/{version}/upload`. This endpoint is also used with external provisioning (not auto-provisioning), where `project_id` and `version` are provided by the client. To prevent anybody from using this endpoint, possibly overwriting or corrupting an existing project, the role `uploader` is required in the default authorization configuration. The role `uploader` is the context should be given with caution, as it provides, at least for the upload process, similar power as an administrator of the instance, since it allows to change data and metadata on any project.

That said, this last endpoint requires the role `uploader`, but going through auto-provisioning and with the first endpoints `POST /projects/upload` and `POST /projects/{project_id}/upload`, the role `creator` is required. Yet both these endpoints will redirect to the one requiring `uploader` permissions. How is it possible for a simple `creator` to be able to call an endpoint reserved for `uploader`? The instance uses an internal pre-signed URL mechanism to temporarily promote the user's role from `creator`to `uploader`, specifically and only for that request, only for the newly provisioned project ID and/or version (the pre-signed upload URL can't be reused for other projects).

TODO: link to "Internal pre-signed URL" section

The uploading process

There are there main steps: preparing the upload, uploading data, and marking the upload as complete.

1. Preparing the upload

The first step is instructing ArtifactDB we want to upload data. As previously seen, we can use different entry endpoints, but overall in the end, they all converge to the same one, `/projects/{project_id}/version/{version}/upload`, also known as the "fully qualified upload endpoint", since project ID and version are fully specified. In other words, the parameters and body content of the request going to that final endpoint are the same for all upload endpoints, the other endpoints are "just" to used during auto-provisioning.

With that in mind, in the endpoint `/projects/{project_id}/version/{version}/upload`, we thus specify the `project_id` as well as the `version` within that project. In the body of that upload request, we instruct ArtifactDB what the files we want to upload, and when we think we wil be done uploading data:

```
{
  "filenames": [
    "report.txt",
    "report.txt.json"
  ],
```

```
  "completed_by": "in 10 minutes"
}
```

Here, we want to upload two files, one containing some data (`report.txt`) and the other containing metadata describing the data, as a json file (`report.txt.json`). The `report.txt` file can contain anything, let's say:

`bash This report is so amazing, wow.`

The actual structure of the JSON document for the metadata depends on the schema used by the ArtifactDB API, and thus is specific to the instance. In our example, we'll use a simple schema named `compiled_report`, with the following structure (again, this schema is just an example, it could be anything… almost):

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "$id": "compiled_report/v1.json",
  "type": "object",
  "title": "Compiled report",
  "description": "A HTML report generated from compilation of
      executable code",
  "required": [
    "source",
    "md5sum",
    "path"
  ],
  "properties": {
    "source": {
      "type": "string",
      "title": "Source report path",
      "description": "A string containing a path to the source file."
    },
    "md5sum": {
      "type": "string",
      "title": "MD5 checksum",
      "description": "A string containing the expected MD5 checksum for
          the compiled report."
    },
    "path": {
      "type": "string",
      "title": "Relative path",
      "description": "A string containing the path to the compiled
          report."
    }
  }
}
```

According that schema, we need to provide fields `source`, `md5sum` and `path`, so the `report.txt.json` could look like this:

```
{
  "source": "report.pl",
  "path": "report.txt",
  "md5sum": "38a3b0a6d8a9b6df7165a7d10cc8a57f",
  "$schema": "compiled_report/v1.json"
}
```

> Note: in the source field, we just pretend the report is generated a from Perl script.

The `$schema` field tells the ArtifactDB API what kind of artifact is being uploaded. It internally translates into an index in Elasticsearch (see architecture for more).

With these two files ready, we can proceed to the upload. We'll pretend to upload to project `PRJ000001` for version 1.

In the rest of this document, we assume `$token` is an environment variable containing a JWT token containing the necessary permissions to upload data (see section above about permissions)

```
$ token=eyJhbGciOiJSUzI1NiIsInR5c...
$ curl -XPOST https://myinstance.mycompany.com/projects/PRJ000001/version/1/upload \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $token" \
    --data '{
        "filenames": [
            "report.txt",
            "report.txt.json"
        ],
        "completed_by": "in 5 minutes"
    }'
```

```
{
    "project_id":"PRJ000001",
    "version":"1",
    "revision":"REVISION-1",
    "presigned_urls":{
        "report.txt":"https://mybucket.s3.amazonaws.com/PRJ000001/1/report.txt?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAVB4ETCFN2Z4X3R73%2F20201222%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20201222T183917Z&X-Amz-Expires=120&X-Amz-SignedHeaders=host&X-Amz-Signature=d2be08f767e313e8d002b54ccc2b590d6783b13cbc3438459ac3270d0e437def",
        "report.txt.json":"https://mybucket.s3.amazonaws.com/PRJ000001/1/report.txt.json?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAVB4ETCFN2Z4X3R73%2F20201222%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20201222T183917Z&X-Amz-Expires=120&X-Amz-SignedHeaders=host&X-Amz-Signature=
```

```
                c2626a62321d211b16cefb79a6443d4f3aea3bfe6cf3bd350ee084dcdbeb4692
                "
        },
        "complete_before":"2020-12-22T19:39:17.388465",
        "completion_url":"/projects/PRJ000001/version/1/complete?revision=
            REVISION-1&purge_job_id=7c3e2d69-d816-4965-b347-116ab9ef47ad",
        "purge_job_id":"7c3e2d69-d816-4965-b347-116ab9ef47ad",
        "expires_job_id":null
```


The response contains several important information: - ResultsDB has assigned the `revision` **REVISION-1**. It's the first time we upload artifact for the project, so this is the first revision. - a `completion_url` is provided, we'll use it during the last step. - we said, during the POST request, we'll be done uploading files in 5 minutes, ResultsDB reports a `completion_before` data, as a reminder (datetimes are in UTC). Should we move past this time limit, our data would be automatically be purged (removed from s3). - finally, and most importantly, we obtain two pre-signed URLs, for each of our report files.

At that stage, the project has entered the state `uploading` (see lifecycle). It's not possible to upload any other data for that project, we would obtain a lock error:


```
{
    "status":"error"
    "reason":"Project 'PRJ000001' is locked: {'stage': 'uploading', '
        info': True}"
}
```


The only way is to move forward and upload files and hit the completion URL, or wait the completion time to expire (in our case, 5 minutes).

Let's move on the next section and upload data.

2. Uploading data

With the pre-signed URLs we obtain during the provisioning step, we can now upload our files. These pre-signed URLs points to Amazon S3, the upload itself doesn't go through the API but rather directly to S3. Using this process, we can benefit from the cloud bandwidth. Let's upload our two files. This time, there's no need for a token, these pre-signed URLs already embeds permissions.

> Note: these pre-signed URLs are only valid for a limited time, 2 minutes… They can be used multiple times during that time frame though.
>
> Content-type for the files must be set by the client, and It can not be change after that. Example: -H "Content-Type: text/plain"

```
$ curl --upload-file report.txt "https://mybucket.s3.amazonaws.com/PRJ000001/1/report.txt?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAVB4ETCFN2Z4X3R73%2F20201222%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20201222T183917Z&X-Amz-Expires=120&X-Amz-SignedHeaders=host&X-Amz-Signature=d2be08f767e313e8d002b54ccc2b590d6783b13cbc3438459ac3270d0e437def"
$ curl --upload-file report.txt.json "https://mybucket.s3.amazonaws.com/PRJ000001/1/report.txt.json?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAVB4ETCFN2Z4X3R73%2F20201222%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20201222T183917Z&X-Amz-Expires=120&X-Amz-SignedHeaders=host&X-Amz-Signature=c2626a62321d211b16cefb79a6443d4f3aea3bfe6cf3bd350ee084dcdbeb4692s"
```

Everything has been uploaded properly, we can now move the final stage.

3. Completing the upload

We can now inform the ArtifactDB API that we're done uploading the files and it should now proceed to their integration. We use the `completion_url`. It's also during that time we can specify permissions for the upload. We'll set a specific owner, and public read access, as well as scope *project*, in the body of the PUT request (refer to artifacts' permissions for more):

```
$ curl -XPUT "https://myinstance.mycompany.com/projects/PRJ000001/version/1/complete?revision=REVISION-1&purge_job_id=de42fb43-974b-4320-afa5-574bdd33a632" \
    -d '{"owners": "lelongs", "write_access": "owners", "read_access": "public", "scope": "project"}' \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $token"
```

```
{
    "status":"accepted",
    "job_url":"/jobs/f31a86b5-1d63-4f0c-a035-8ebd0abc684b",
    "job_id":"f31a86b5-1d63-4f0c-a035-8ebd0abc684b"
}
```

The response code is 202 (not shown here) meaning the request has been accepted, ArtifactDB is processing it in an asynchronous manner. In return, we obtain information about that asynchronous job. We can use the `job_url` to check the progress:

> Note: there's no need to pass a token for the `/jobs` endpoints, as an randomly generated ID is required as a job ID, it's basically not possible to "guess" it. Also, there's no sensitive information displayed by that endpoint.

```
$ curl "https://myinstance.mycompany.com/jobs/f31a86b5-1d63-4f0c-a035-8
   ebd0abc684b"
```

```
{
    "status":"SUCCESS",
    "result":{
        "project_id":"PRJ000001",
        "indexed_files":1
    },
    "traceback":null,
    "children":[],
    "date_done":"2020-12-22T18:56:07.962614",
    "task_id":"f31a86b5-1d63-4f0c-a035-8ebd0abc684b"
}
```

It's a `SUCCESS`, we can see our project has been integrated, and one file has indexed (the metadata).

Since the project is public, we can easily retrieve metadata for instance, without any token:

```
$ curl "https://myinstance.mycompany.com/projects/PRJ000001/version/1/
   metadata"
```

```
{
    "results":[
        {
            "source":"report.pl",
            "path":"report.txt",
            "md5sum":"38a3b0a6d8a9b6df7165a7d10cc8a57f",
            "_extra":{
                "project_id":"PRJ000001",
                "metapath":"report.txt.json",
                "version":"1",
                "meta_uploaded":"2020-12-22T18:54:53+00:00",
                "uploaded":"2020-12-22T18:54:53+00:00",
                "file_size":141,
                "revision":"REVISION-1",
                "numerical_revision":1,
                "permissions":{
                    "scope":"project",
                    "owners":[
                        "lelongs"
                    ],
                    "read_access":"public",
                    "write_access":"owners"
                },
                "type":"compiled report",
                "id":"PRJ000001:report.txt@1",
                "$schema":"compiled_report/v1.json",
                "meta_indexed":"2020-12-22T19:17:57.748477+00:00",
                "index_name":"resultsdb-prd-default-20200817"
```

```
            }
        }
    ],
    "count":1,
    "total":1
}
```

Please refer to access for more about fetch and searching data from ArtifactDB.

Uploading using STS credentials

AWS Security Token Service allows to access cloud resources in a native way, using standard AWS Access Key ID, Secret Access Key and a Session Token. These credentials are temporary and can be used with any of the AWS tools, such `awscli`. ArtifactDB can provide such credentials for the upload process, when the instance has access to a STS credential provider (if that's not the case, the following procedure will fail). Using STS credentials allows to parallel uploads, multi-part uploads with auto-retry, and overcome the limit of 5GiB per file (inherent to pre-signed URLs), reaching the S3 limit of 5TB per file. Finally, using STS credentials allows to upload folders containing lots of files without having to provision one pre-signed URL per file.

The upload procedure itself is very similar to what has been described using pre-signed URLs. The main different is the upload `mode` parameter used in one of the `/upload` endpoint. By default, if not specified, the value is `s3-presigned-url`, setting it to `sts-credentials` triggers to generation of STS credentials:

```
$ curl -XPOST https://myinstance.mycompany.com/projects/PRJ000001/
    upload \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $token" \
    --data '{
        "filenames": [
            "report.txt",
            "report.txt.json"
        ],
        "completed_by": "in 5 minutes",
        "mode": "sts-credentials",
    }'
```

The response contain an empty `presigned_urls`, and a new `sts` section:.

```
{
    "project_id":"PRJ000001",
    "version":"3",
    "revision":"NUM-3",
    "presigned_urls":{},
    "sts":{
```

```
        "credentials":{
            "aws_access_key_id":"ASIAVB...",
            "aws_secret_access_key":"PsXVG...",
            "aws_session_token":"FwoGZXIvYXdz..."
        },
        "session_name":"almighty-session-tmp-dev-ho1VFbp0",
        "expiration":"2022-12-20T20:06:04+00:00",
        "bucket":"myapi-bucket",
        "prefix":"PRJ000001/3/"
    },
    "links":{},
    "complete_before":"2022-12-20T20:01:02.468069+00:00",
    "completion_url":"/projects/PRJ000001/version/3/complete?revision=
        NUM-3&purge_job_id=e3dc4b2f-dc74-4d45-88b4-225a1f8daac7&X-ADB-
        Credential=cSOQPky1EXtqEr...",
    "abort_url":"/projects/PRJ000001/version/3/abort?purge_job_id=
        e3dc4b2f-dc74-4d45-88b4-225a1f8daac7&X-ADB-Credential=
        uXWsxh3R32f...",
    "purge_job_id":"e3dc4b2f-dc74-4d45-88b4-225a1f8daac7",
    "expires_in":null,
    "expires_job_id":null
```

Within that `sts` section:

- `credentials` contains the actual AWS credentials, ready to be used. Note the format matches boto3 signatures so the dict can be passed as python kwargs, as shown below.
- `session_name` is an session ID, for informational/troubleshooting purpose. In this case, there's also a hint about the STS credential provider used, `Almighty`, a security service part of the ArtifactDB platform.
- `expiration` informs when the credentials will expire. Note this expiration time cannot be more than 12h, due the AWS STS service itself.
- `bucket` and `prefix` specifies what the bucket, and folder within that bucket, these credentials are valid for. Indeed, the STS provider (Almighty) generates credentials for a given AWS resource. Here, we want to upload data to project `PRJ000001`, for a new version provisioned as number `3`. The credentials should only be valid this specific operation, and nothing else.

An important point is STS credentials can be used independently from the `links` URLs. If links are specified in the upload payload, these links URLs will be returned, along with the STS credentials, and should be call just as seen before.

How to use these credentials? As mentioned, any AWS standard tools using access and secret keys can be used with thesee. Here are 2 examples, with bash and `awscli`, and with `python` and `boto3`. Note if errors like "An error occurred (AccessDenied)" are raised, it means the credentials have expired, or were not valid in the first place, for instance because the STS provider doesn't cover the AWS resources used by the instance.

Using bash, awscli and curl[1]:

```
# awscli can be used with eg. standard environment variables:
export AWS_ACCESS_KEY_ID="ASIAVB..."
export AWS_SECRET_ACCESS_KEY="PsXVG.."
export AWS_SESSION_TOKEN="FwoGZXIvYXdz..."  # don't forget the session
    token
# then
aws s3 ls s3://myapi-bucket/PRJ000001/3/  # we get a listing
# but
aws s3 ls s3://myapi-bucket/PRJ000001/2/  # access denied, not allowed
    because version doesn't match
aws s3 ls s3://myapi-bucket/PRJ000006  # same, project doesn't match
aws s3 ls s3://myotherapi-bucket  # same, bucket doesn't match
# let's upload some data
aws s3 sync my_staging_dir s3://myapi-bucket/PRJ000001/3/
# complete the upload, with the `completion_url`
curl -XPUT https://myinstance.mycompany.com/projects/PRJ000001/version
    /3/complete?revision=NUM-3&purge_job_id=e3dc4b2f-dc74-4d45-88b4-225a
    ...
```

Using python and boto3 is pretty similar:

```
import requests, boto3, glob, pathlib
# assuming `response` comes from a `requests.post(".../upload")` call
bucket = response.json()["sts"]["bucket"]
prefix = response.json()["sts"]["prefix"]
creds = response.json()["sts"]["credentials"]
# `creds` keys match what boto3 wants, how convenient...
s3client = boto3.client("s3",**creds)
staging_dir = "my_staging_dir"
for filename in glob.iglob(f"{staging_dir}/**"):
    key = "{}{}".format(prefix,pathlib.Path(filename).relative_to(
        staging_dir))
    s3client.upload_file(filename,bucket,key)

# complete the upload
requests.put("https://myinstance.mycompany.com{}".format(response.json
    ()["completion_url"]))
```

Another option would be to use one of the generic ArtifactDB clients, which transparently handles all these steps, but this is outside of the scope of this document, which goal is to show low level API interactions.

[1] These roles are coming from the tokens (JWT), see authentatication for more.

Transient artifact

It is also possible to upload transient artifacts, which automatically get deleted after a certain amount of time. This can be specified while hitting one of the /upload endpoint, by specify the expires_in field. The format is the same as the completed_by, eg. in 2 days, or dates like 2020-12-31 12:59:59.

> Note: expires_in must happen **after** completed_by.

```
$ curl -XPOST https://myinstance.mycompany.com/projects/upload \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $token" \
    --data '{
        "filenames": [
            "report.txt",
            "report.txt.json"
        ],
        "completed_by": "in 1 minute",
        "expires_in": "in 2 minutes"
    }'
```


```
{
    "project_id":"PRJ000001",
    "version":"1",
    "revision":"REVISION-1",
    "presigned_urls":{
        "report.txt":"https://mybucket.s3.amazonaws.com/PRJ000001/1/report.txt?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAVB4ETCFN2Z4X3R73%2F20201222%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20201222T193126Z&X-Amz-Expires=120&X-Amz-SignedHeaders=host&X-Amz-Signature=d8fc4e7301375ac0a26617f56efbe431e1f3f58b568bf746f35b63eaeba3f924",
        "report.txt.json":"https://mybucket.s3.amazonaws.com/PRJ000001/1/report.txt.json?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAVB4ETCFN2Z4X3R73%2F20201222%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20201222T193126Z&X-Amz-Expires=120&X-Amz-SignedHeaders=host&X-Amz-Signature=2935c706ae915deaac1723d4d033c5430fe6f131894aa06d6c715662a13128da"
    },
    "complete_before":"2020-12-22T19:32:26.641375",
    "completion_url":"/projects/PRJ000001/version/1/complete?revision=REVISION-1&purge_job_id=ba40f466-347f-4c7c-9cc2-db85c92db0c2",
    "purge_job_id":"ba40f466-347f-4c7c-9cc2-db85c92db0c2",
    "expires_job_id":"34229a07-2e21-480d-9466-fcd8ff257f74"
}
```


The same kind of information as before is returned, with an additional expires_job_id. Internally, ArtifactDB scheduled a job with this ID, which will run in 2 minutes (a little bit less now), to purge the data from the API.

After two long minutes, if we query again the data, we get an error:

```
$ curl "https://myinstance.mycompany.com/projects/PRJ000001/version/1/
   metadata"
```

```
{
    "status":"error"
    "reason":"No such project/version"
}
```

As expected, the project has been removed from ArtifactDB instance. The files on Amazon S3 have been deleted, and the documents from Elasticsearch have been removed as well. It's gone…

> Note: if permissions were defined with scope `project`, these permissions are kept on S3. It means if you had upload permissions before for that project, you still have them after the purge has occured, which means you still have permissions to upload a new version for that project. In other words, you still own that space…

Linking data (deduplication)

One common scenario when uploading artifacts is metadata itself needs an update, eg. with additional fields describing the underlying data in more details, but the data doesn't change. As previously seen, using this new metadata will end up in the creation of new version of project which, without further consideration, would contain the exact same data file(s) as the previous one. Blatant data duplication, waste of storage and money.

Data file duplication can be addressed by using ArtifactDB links. They work in a very similar way as symlinks on a POSIX filesystem: ArtifactDB can create a link pointing to another ArtifactDB ID. In the example, a link in version 2 would point to the data file in version 1. This would happen within the same project, but ArtifactDB links can also be used to refer to files in other projects ("cross-projects" links)[2]. An important point to remember linking metadata is currently **not** supported, only data files can be linked[3].

[2] Linking to another ArtifactDB instance is also a possibily (on the roadmpa), using the GPRN notation. This comes with a risk if the targetted file disappear on the other instance, for some reason.

[3] though there can be duplication in metadata files, they are usually much smaller than data files, so the waste of storage is less important. What is more important there is the maintenance of the instance. Often an admin may need to open a metadata file for troubleshooting purposes (eg. that metadata file isn't indexed for some reason), having linked metadata would make that operation more complex, by manually resolving links. We loose a bit of storage in exchange of a more maintainable system.

The linking mecanism is currently initiated from the client itself. Though on the roadmap, there is no automatic data deduplication happening on the backend side, which would inspects data files, find duplicated ones and automatically create links. The client itself needs to know and provide linking information to the API. There are different ways to achieve this: by providing ArtifactDB IDs directly or providing md5sum information [4]

Linking data with explicit ArtifactDB IDs

The first way to create links is to explicitly specify that a given filename (ie. artifact) should be link to an ArtifactDB ID. On the previous example, assuming we create a new version 2 where we know (as the client) that the data file `RES000041637_1_MAE.hdf5` didn't change, we could instruct the API to link that file as such:

```
$ curl -XPOST https://myinstance.mycompany.com/projects/PRJ000001/
    upload \
    -H "Content-Type: application/json" \
    -H "Authorization: Bearer $token" \
    --data '{
        "filenames": [
            "RES000041637_1_MAE.hdf5.json",
            {
                "filenames": {
                    "filename": "RES000041637_1_MAE.hdf5",
                    "check":"link",
                    "value":{
                        "artifactdb_id": "PRJ000001:RES000041637_1_MAE.
                            hdf5@1"
                    }
                }
        ],
        "completed_by": "in 5 minutes"
    }'
```

Note the list `filenames` now contains not only a string representing the metadata file, but also a dictionary structure describing the link instructions, with the following keys:

- `filename` is used to produce the ID for the link (which would be `PRJ000001:RES000041637_1_MAE.hdf5@2` after the version 2 is created)
- `check: link` refers to both the linking mechanism to use and the check to perform for that. In this case, we specify that we want link to an ArtifactDB ID, a verification will be performed to verify the linked data exists.

[4] Another possibitly that might be implemented in the future in using file timestamps such as "mdtm" (modified datetime) and file size, though these don't provide good indication of uniqueness.

- `value` is dict structure, here representing the ArtifactDB ID we want to link to (same file but in version 1 of the same project, thus `PRJ000001:RES000041637_1_MAE.hdf5@1`)

The instance, when receiving this payload, checks that the target file `PRJ000001:RES000041637_1_MAE.hdf5@1` exists. If not, a HTTP 400 error is returned. If the payload is valid, the response looks similar as previously seen, with the additional `links` key:

```
...
    "links":{
        "RES000041637_1_MAE.hdf5":"https://myinstance.mycompany.com/
            link/dGVzdC1PTEE...LmhkZjVAOQ==/to/dGVzdC1PTEEwMDA....
            LmhkZjVAMg==?X-ADB-Credential=sFG9vDWY...vUVneOr9iI"
        },
...
```

The URL served in this section works the same as a pre-signed URL, using `PUT` verb. If any data is provided in the body, it is ignored. This is not recommended though, as it would slow down the process of creating these links.

```
$ curl -XPUT https://myinstance.mycompany.com/link/dGVzdC1PTEE...
    LmhkZjVAOQ==/to/dGVzdC1PTEEwMDA....LmhkZjVAMg==?X-ADB-Credential=
    sFG9vDWY...vUVneOr9iI
```

This indeed uses the REST endpoint `/link/{b64source}/to/(b64target}`, where `{b64source}` and `{b64target}` are the base64 encoded ArtifactDB ID of respectively the source and target file. This endpoint is reserved for `admin` role by default, thus the presence of the pre-signed authentication information under the parameter `X-ADB-Credential` (see TODO: link to "Internal pre-signed URL" section).

Linking data using MD5 checksum

Linking data can also be done by providing MD5 checksum information. As opposed to providing explicit ArtifactDB ID links, using MD5 checksum allows to link data *only* within a project. This restriction is mostly there to avoid hash collisions. Other checksums are not currently supported, should that happen and mitigate this risk, that restriction may disappear in the future.

The instance itself doesn't compute a MD5 checksum of the data it receives, but rather rely on metadata information on that regard. This means the checksum information may appear at different places depending on the schemas defined for the instance. Not only the checksum must be provided, but the field in which it should be found in the targetted file:

```
$ curl -XPOST https://myinstance.mycompany.com/projects/PRJ000001/
    upload \
    -H "Content-Type: application/json" \
```

```
    -H "Authorization: Bearer $token" \
    --data '{
        "filenames": [
            "RES000041637_1_MAE.hdf5.json",
            {
                "filenames": {
                    "filename": "RES000041637_1_MAE.hdf5",
                    "check":"md5",
                    "value":{
                        "field": "md5sum",
                        "md5sum": "a0de57f771efee53465bfaccc8eac519"
                    }
                }
        ],
        "completed_by": "in 5 minutes"
    }'
```

Specifically, `field` contains the name of metadata field that should hold the checksum value found in `md5sum`. In this example, we're expecting a metadata document like this:

```
{
    "$schema": "whatever/v1.json",
    ...
    "some_other_field": "blah",
    ...
    "path": "RES000041637_1_MAE.hdf5",
    "md5sum": "a0de57f771efee53465bfaccc8eac519",
    "_extra": {
        "project_id": "PRJ000001",
        "version": "1",
        "id": "PRJ000001:RES000041637_1_MAE.hdf5@1",
        ...
    }
}
```

Within metadata documents part of the project (`_extra.project_id:PRJ000001`), we're looking for a match with `md5sum:a0de57f771efee53465bfaccc8eac519`). If the checksum field is nested, the so-called "dot-field" notation can be used, eg. `some_root_key.md5um`.

Permissions

There are two distinct types of permissions involved in the process of uploading artifacts; permissions used to access API endpoints, and permissions defined at the data level.

First, accessing the different endpoints requires different roles[5] or permissions:

[5]These roles are coming from the tokens (JWT), see authentatication for more.

- roles `creator` or `uploader` is required to access the `POST /projects/upload` endpoint, that is, to initiate the creation of a new project.
- role `uploader` may be required to access the `POST /projects/{project_id}/upload`, that is, to create a new version within an existing project. If a user has `write_access` permissions for the project (data-level permssions), usually corresponding to the `owner` role, this role is not required, user has ownership over the project and should be allowed to create a new version.
- role `uploader` is required the `POST /projects/{project_id}/version/{version}/upload` endpoint.

The role `uploader` should be reserved for service account only, or when identifers or versions are provisioned externally, as previously explained. This role has the capacity to overwrite data in existing projects and versions and can be seen as "data upload adminitrator" role. The role `creator` on the other hand is aimed at end-users and can be assigned for instance to all authenticated users, or a subset, depending on configuration of the authentication service.

Re-indexing

Re-indexing happens during upgrade and/or maintainance operations. This requires admin privileges, as this process is available with the endpoint `/index/build`. During a global re-indexing, all the metadata is pulled from the storage and sent to Elasticsearch. Depending on the size of the ArtifactDB instance and its metadata, this step can take several hours.

Concurrent re-indexing is not allowed: the API puts a global lock in place, named `__all_projects__`, with a stage `index_all`. The re-indexing needs to be completed, either with a success or failure, to release that lock before a new re-indexing can be triggered. During that time, the endpoint `/index/status` will report `state: re-indexing` in the response. Once done, that state can turn into `ok` (success) or `failed`. This state can be used to decide when to update aliases (if using them) from old indices, to new freshly populated indices.

Note during a re-indexing, new projects, or new versions of existing projects can be added (with locking mechanism specific to the projects applying, as explained above). Re-indexing *existing* metadata from a storage and adding *new* metadata are two distinct processes, independent from each other. Namely, they don't have the same impact in terms of operational process, when new indices are being populated in the backend, while the frontend REST API still serves existing metadata from the old indices (this is the main use case of re-indexing).

TODO: explain the content of `_extra` key

TODO: explain fetch endpoint (/files)

TODO: explain search endpoint and aggs

Exploring GPRNs

The identifiers section describe what a GPRN is, while this section describes how to use `/gprns` endpoints available in ArtifactDB APIs.

Validating a GPRN

As a utility or helper, the endpoint `GET /gprn/{gprn}/validate` can be used to make sure a GPRN is valid. Few examples:

```
# valid GPRN
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::artifact:
    PRJ00000022:experiment-1/assay-2.h5@3/validate
```

```
{
    "status": "ok"
}
```

```
# invalid GPRN (`type-id` is `artifact` but should be `project`,
    because the `resource-id` is not an ArtifactDB ID)
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::artifact:
    PRJ00000022/validate
```

```
{
    "status":"error",
    "reason":"unable to parse ID"
}
```

```
# incorrect environment (the GRPN refers to MyAPI "production", but the
     instance serves MyAPI "development")
$ curl https://dev.myapi.mycompany.com/gprn/gprn::myapi::project:
    PRJ00000022/validate
```

```
{
    "status":"error",
    "reason":"Invalid 'environment'"
}
```

Validating a GPRN doesn't require authentication, as this only operation involved is the parsing (there's no underlying queries happening).

Locating a GPRN

Since ArtifactDB APIs stores data on AWS S3 any artifact, project and version, or just a project, corresponds to a specific S3 URL and ARN. The endpoint `GET /gprn/{gprn}/locate` provides such locations. Obtaining the location of a GPRN requires permissions to access the GPRN in the first place, this endpoint is thus under authentication.

The following example shows how to determine the location of different GPRNs.

```
# Locating a HDF5 file, part of a dataset
# GPRN is: gprn:dev:myapi::artifact:PRJ00000022:experiment-1/assay-2.h5@3
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::artifact:PRJ00000022:experiment-1/assay-2.h5@3/locate
```

```
{
  "s3_url": "s3://my-bucket/PRJ00000022/3/experiment-1/assay-2.h5",
  "s3_arn": "arn:aws:s3:::my-bucket/PRJ00000022/3/experiment-1/assay-2.h5"
}
```

```
# Locating a project+version folder
# GPRN is: gprn:dev:myapi::project:PRJ00000022@3
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::project:PRJ00000022@3/locate
```

```
{
  "s3_url": "s3://my-bucket/PRJ00000022/3/",
  "s3_arn": "arn:aws:s3:::my-bucket/PRJ00000022/3/"
}
```

Note a "revision" (eg. `NUM-3`) can also be passed as version information in the GPRN, this locate endpoint will resolve the "revision" to the actual "version", corresponding to an existing folder on S3.

By default, existence of S3 keys is verified, this can be bypassed by specifying the query string parameter `?check=false` (useful in some use cases involving provisioning).

Obtaining parents lineage

Within an ArtifactDB API, an artifact belongs to a version, which belongs to a project. This lineage information can be obtained using the endpoint `GET /grpn/{gprn}/parents`.

```
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::artifact:PRJ00000022:experiment-1/assay-2.h5@3/parents
```

```
[
```

```
    {
        "type":"version",
        "gprn":"gprn:dev:myapi::project:PRJ00000022@3"
    },
    {
        "type":"project",
        "gprn":"gprn:dev:myapi::project:PRJ00000022"
    },
    {
        "type":"projects",
        "gprn":"gprn:dev:myapi::project"
    },
    {
        "type":"service",
        "gprn":"gprn:dev:myapi"
    }
]
```

This endpoint doesn't require authentication, as the process only involves parsing the GPRN.

Obtaining children lineage

In the same way as parents lineage, given a GPRN, its children can be obtained using the endpoint `GET /gprn/{gprn}/children`. This endpoint may require authentication in order to query and determine children. Also, the maximum number of children is currently limited to 250 by default, should this limit reached, a `partial_results: true` would be present in the response.

```
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::project:
    PRJ00000043/children
```

```
{
  "children": [
    "gprn:dev:myapi::artifact:PRJ00000043:dataset.json@1",
    "gprn:dev:myapi::artifact:PRJ00000043:experiment-1/assay-1.hdf5@1",
    "gprn:dev:myapi::artifact:PRJ00000043:experiment-1/assay-2.hdf5@1",
    "gprn:dev:myapi::artifact:PRJ00000043:experiment-1/coldata/simple.
        csv@1",
    "gprn:dev:myapi::artifact:PRJ00000043:experiment-1/experiment.
        json@1",
    "gprn:dev:myapi::artifact:PRJ00000043:experiment-1/rowdata/simple.
        csv@1",
    "gprn:dev:myapi::artifact:PRJ00000043:sample_data/simple.csv@1",
    "gprn:dev:myapi::artifact:PRJ00000043:sample_mapping.csv@1"
  ]
}
```

Note it's not possible to request children of a GPRN "higher" than one pointing to a specific project, eg. trying to obtain children of all projects will return an error:

```
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::project/
    children
```

```
{
  "status": "error",
  "reason": "Requesting children without a resource-id component is not
      allowed (for now)"
}
```

Checking permissions

In order to know if a given user (through his/her JWT token), the endpoint `GET /gprn/{gprn}/permissions` can be used. It returns a `HTTP 200` if allowed, or `HTTP 404 Not Found` if the user isn't allowed to access it or if the GPRN doesn't exist within the API.

Note: the status code 404 "Not Found" is returned instead of a 403 "Not authorized". This is by design: an ArtifactDB API doesn't reveal whether an artifact exists if the requester isn't allowed to access it. This logic is somewhat similar to a firewall rule dropping packets (we don't know if the target exists) instead of rejecting them (we know the target exists but we get a explicit deny, which is informative on its own).

```
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::project:
    PRJ00000043/permissions
```

```
{
  "allowed": true
}
```

When not allowed (or the GPRN doesn't exist):

```
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::project:
    DS000006849/permissions
```

```
{
  "status": "error",
  "reason": "No such GPRN"
}
```

Checking permissions on a GPRN "higher" than a specific project is not allowed:

```
$ curl https://dev.myapi.mycompany.com/gprn/gprn:dev:myapi::project/
    permissions
```

```
{
  "status": "error",
```

```
  "reason": "Requesting permissions without a resource-id component is not allowed"
}
```

TODO: explain project management endpoints

TODO: explain index management endpoints

TODO: explain permissions management endpoints

TODO: explain schema endpoints

TODO: explain sequence management endpoints

TODO: explain tasks and plugins endpoints

TODO: explain config endpoint

Using the administration terminal

TODO: enable/disable terminal, maintainer operator, create admin users, wetty URL to connect, loading tools/admin.py, examples, permissions

An administration terminal can optionally be deployed and used to access the ArtifactDB instance's internal. This terminal exposes a pod shell from within the Kubernetes namespace. That pod contains the exacts same code being used by the instance itself. A convenient script named `tools/admin.py` can be used to load and instantiate major components of the instance, notably a backend manager instance, from which pretty much every operations are possible.

Accessing the terminal

When enabled and active, the terminal can be access under the path prefix `/shell`, with any web browser. A Linux login page is then displayed, asking for a username and password. Administration users need to be created first, either when installing/upgrading the instance by providing a secret containing that list of users (recommended, part of the deployment code), and after the deployment, by manually populating the secret, directly on the cluster (not recommended, for temporary access setting only). See below on how to create these users.

Once logged, we land in bash terminal, within the admin pod living in the Kubernetes cluster.

Figure 0.1: Terminal login

Once logged, we can run the `tools/admin.py` script. This python script is present in all ArtifactDB instances, loads commonly used libraries and instantiates major components. After running and displaying a horrendous (yet interesting) amount of logs, the environement is loaded and a special variable `mgr` is available for us, reprensent a central component, a backend manager instance.

```
$ ipython
In [1]: run tools/admin.py
Using config files: ['./etc/config-uat.yml']
...
...
INFO:root:Aliases found (index => alias): {'olympusapi1-uat-v1-20221117
    ': 'olympusapi1-uat-v1'}
INFO:root:Aliases found (index => alias): {'olympusapi1-uat-v1-20221117
    ': 'olympusapi1-uat-v1'}

In [2]: mgr
Out[2]: <backend.manager.BackendManager at 0x7f25d1755400>
```

From there, we can access all the manager's sub-components and interact with ArtifactDB, using python code and the framework itself. Notably, `mgr.es` points to the ElasticSearch manager handling index queries, while `mgr.s3` deals with data storage in general. You will find some examples throughout this documentation, using these admin pod, terminal and `mgr` instance, as a conventional way to achieve advanced operations on ArtifactDB instances.

As a usage illustration, in the below example, we list all projects available from the storage, and index one of them.

Activating/deactivating the terminal

```
# Assuming `tok` contains an admin token
> requests.put(
    url + "/maintenance/requests",
    json={"name": "scale-deployment", "args": ["admin",1]},
    headers={"Authorization":f"Bearer {tok}"}
)
<Response [200]>
# checking maintenance requests (before it's processed by the operator)
> requests.get(
    url + "/maintenance/status",
    headers={"Authorization":f"Bearer {tok}"}
).json()
{'state': None,
 'requests': [{'stage': None,
   'info': {'name': 'scale-deployment', 'args': ['admin', 1], 'kwargs':
        None},
   'created': '2023-01-11T22:05:09.524265',
   'owner': 'lelongs'}],
```

```
In [7]: mgr
Out[7]: <backend.manager.BackendManager at 0x7f25d1755400>

In [8]: pids = list(mgr.s3.list_projects())

In [9]: pids
Out[9]: ['OLA000000001', 'OLA000000002', 'OLA000000003']

In [10]: mgr.index_project("OLA000000002")
INFO:root:Global lock acquired (<redis.lock.Lock object at 0x7f25cd9ef610>)
DEBUG:root:Locking project 'OLA000000002' (lock info: {'stage': 'indexing', 'info': {'version': None, 'revision': None}, 'creat
ed': '2023-01-11T23:22:56.706429', 'owner': None})
INFO:root:Global lock released (<redis.lock.Lock object at 0x7f25cd9ef610>)
INFO:root:Found internal metadata: OLA000000002/1/..meta/revision.json => {'revision': 'NUM-1', 'numerical_revision': 1}
DEBUG:root:Inspecting OLA000000002/1/RES000041637_1_MAE.hdf5 with <artifactdb.backend.inspectors.core.ListerInspector object at
 0x7f25cd29f580>
DEBUG:root:Inspecting OLA000000002/1/RES000041637_1_MAE.parquet with <artifactdb.backend.inspectors.core.ListerInspector object
 at 0x7f25cd29f580>
DEBUG:root:While inspecting, generated 2 metadata file(s)
DEBUG:root:Uploading OLA000000002/1/.artifactdb/RES000041637_1_MAE.hdf5.json
DEBUG:root:Uploading OLA000000002/1/.artifactdb/RES000041637_1_MAE.parquet.json
DEBUG:root:Skip OLA000000002/1/..meta/revision.json (internal metadata)
DEBUG:root:Found metadata file: OLA000000002/1/.artifactdb/RES000041637_1_MAE.hdf5.json
DEBUG:root:Found metadata file: OLA000000002/1/.artifactdb/RES000041637_1_MAE.parquet.json
DEBUG:root:Found metadata file: OLA000000002/1/RES000041637_1_MAE.hdf5.json
INFO:root:Register internal metadata: OLA000000002/1/..meta/revision.json => <artifactdb.backend.components.revisions.Numerical
Revision: NUM-1 [1]>
INFO:root:Found internal metadata: OLA000000002/..meta/permissions.json => {'scope': 'project', 'owners': ['lelongs'], 'read_ac
cess': 'viewers', 'write_access': 'owners'}
INFO:root:Found project-specific permissions for OLA000000002
INFO:root:For OLA000000002/1, found permissions <artifactdb.backend.components.permissions.StandardPermissionsWrapper: {'scope'
: 'project', 'owners': ['lelongs'], 'read_access': 'viewers', 'write_access': 'owners'}>
INFO:root:No jsondiff files found
WARNING:root:No schema client found to handle document with $schema entry.minimal/v1.json, this is unexpected
WARNING:root:No specific ES client found for routing documents, using default one
INFO:root:Preparing indexing id=OLA000000002:RES000041637_1_MAE.hdf5@1 [index=olympusapi1-uat-v1-20221117,doc_class=<class 'art
ifactdb.db.elastic.models.ArtifactDBDocumentMinimal'>]
WARNING:root:No schema client found to handle document with $schema entry.minimal/v1.json, this is unexpected
WARNING:root:No specific ES client found for routing documents, using default one
INFO:root:Preparing indexing id=OLA000000002:RES000041637_1_MAE.parquet@1 [index=olympusapi1-uat-v1-20221117,doc_class=<class '
artifactdb.db.elastic.models.ArtifactDBDocumentMinimal'>]
INFO:root:Preparing indexing id=OLA000000002:RES000041637_1_MAE.hdf5@1 [index=olympusapi1-uat-v1-20221117,doc_class=<class 'art
ifactdb.db.elastic.models.ArtifactDBDocumentMinimal'>]
INFO:root:Client <artifactdb.db.elastic.client.ElasticClient: https://dev-elasticsearch.genomics.roche.com/olympusapi1-uat-v1-2
0221117>, indexing 3 documents
DEBUG:root:Bulk index, batch contains 3 documents
INFO:root:Global lock acquired (<redis.lock.Lock object at 0x7f25cd9ef610>)
INFO:root:Releasing project 'OLA000000002' (force=True)
INFO:root:Global lock released (<redis.lock.Lock object at 0x7f25cd9ef610>)
Out[10]: 3
```

Figure 0.2: Example: indexing a project

```
 'started_at': None,
 'stage': None,
 'owner': None,
 'info': {}}
# After a while, request was processed
> requests.get(
    client._url + "/maintenance/status",
    headers={"Authorization":f"Bearer {tok}"}
).json()
{'state': None,
 'requests': [],
 'started_at': None,
 'stage': None,
 'owner': None,
 'info': {}}
```

The terminal becomes accessible at `[url]/shell`.

To disable the terminal, we scale it down to 0

```
> requests.put(
    url + "/maintenance/requests",
    json={"name": "scale-deployment", "args": ["admin",0]},
    headers={"Authorization":f"Bearer {tok}"}
)
```

Accessing the terminal again will result in a “404 Not Found” error.

Design

Architecture

The architecture of an ArtifactDB instance is pretty simple:

1. Data, metadata, permissions (and any other administrative metadata) are stored on AWS S3, considered the source of truth.
2. Metadata is indexed on Elasticsearch, using asynchronous tasks orchestrated by a backend running Celery. Permissions are also injected in the index to implement authorization.
3. A frontend REST API provides multiple endpoints to authenticate and authorize users, fetch, search metadata, orchestrate the storage and retrieval of data files from S3. Data on S3 cannot be accessed unless the corresponding metadata is accessible.

These distributed components act a whole to a provide a scalable backend system. The REST API is implemented using FastAPI, a python framework dedicated to this type of API. A frontend instance, ie. FastAPI processes, is stateless which makes it easy to scale, by just created more and more instances handling the traffic. AWS S3 is a object-store, highly available and reliable.

Celery is used to orchestrate asynchronous backend tasks, like indexing metadata into Elasticsearch. Celery is organized into a broker (RabbitMQ) handling the messaging and possibly queue priorities, a results backend (Redis by default, but also Elasticsearch), a Cebery beat handling task periodic scheduling, and one or more Celery workers, on which tasks are sent and processed. These workers are easily scalable too.

Redis also plays an important role, regardless of Celery, by providing distributed locks (to prevent a client to modify the same project/version at the same time for instance) and several caches to the API (schemas, user authentication information such as AD groups membership and more, public keys from OpenID Connect servers, backend remote tasks definition, custom Elasticsearch scrollers, …). This improves the stateless of the other components to allow easy scaling up and down.

Finally, while in many cases the possible bottleneck could be the database itself, Elasticsearch is a distributed database, also easy to scale by deployment more nodes to the cluster. This is obviously a limit to this, but it’s not rare to multi billions documents clusters (we’re far from this number). The

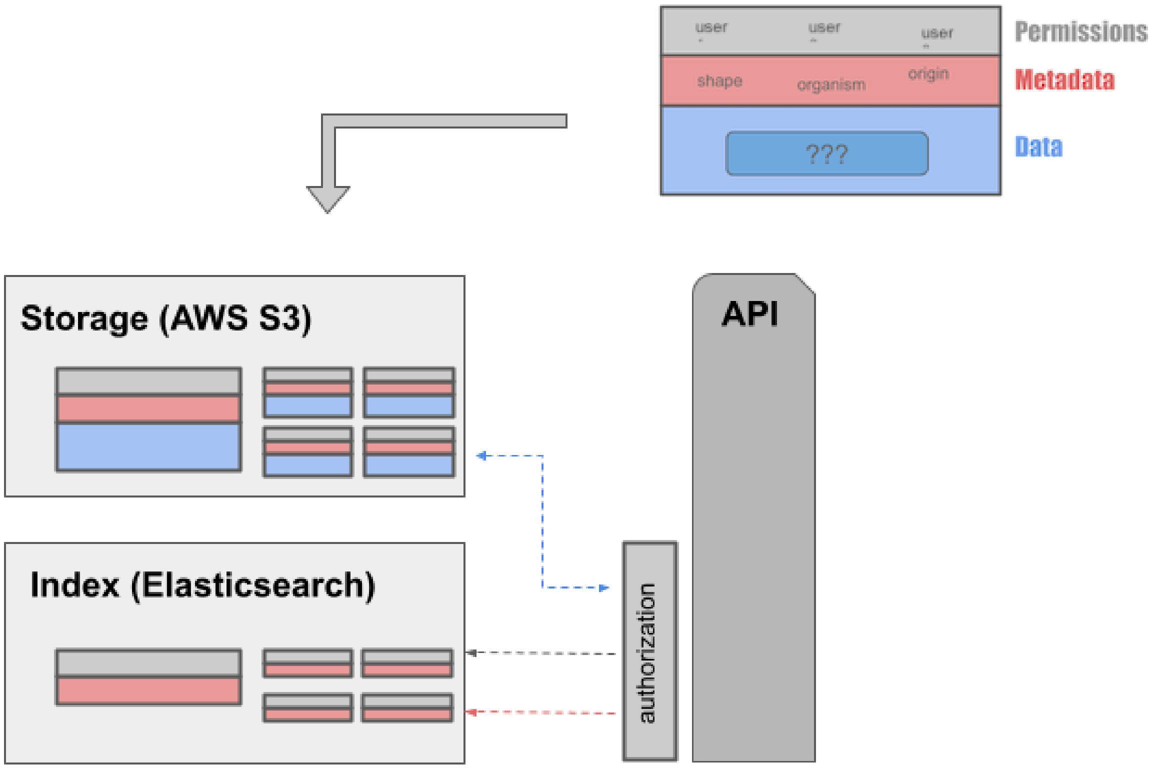

Figure 0.1: ArtifactDB (over-simplified) architecture

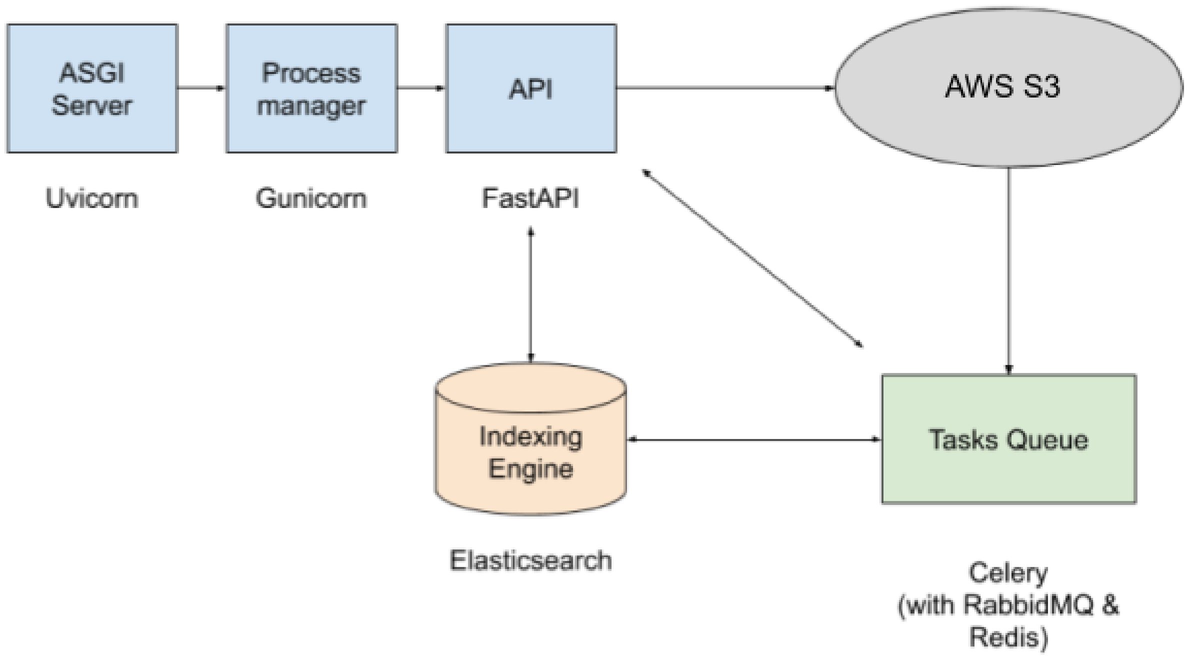

Figure 0.2: Backend components

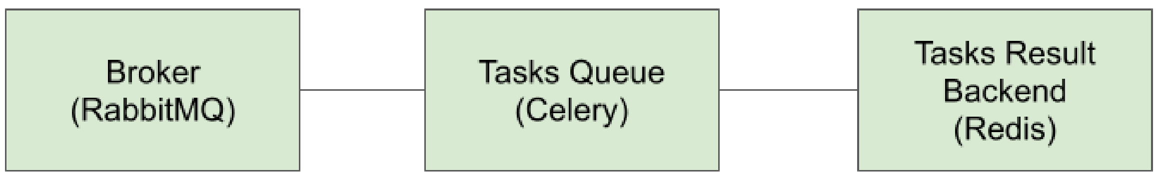

Figure 0.3: Celery tasks queue

queries themselves can be costly though, specially when performing aggregations, but that's pretty much the case for all databases (eg. joins in RDBMS).

ArtifactDB APIs are preferrably deployed on a Kubernetes cluster, to benefit from its ecosystem, resilience, self-healing features. Frontend and backend pods are scaled easily and accordingly (though there's no dynamic scaling at the moment).

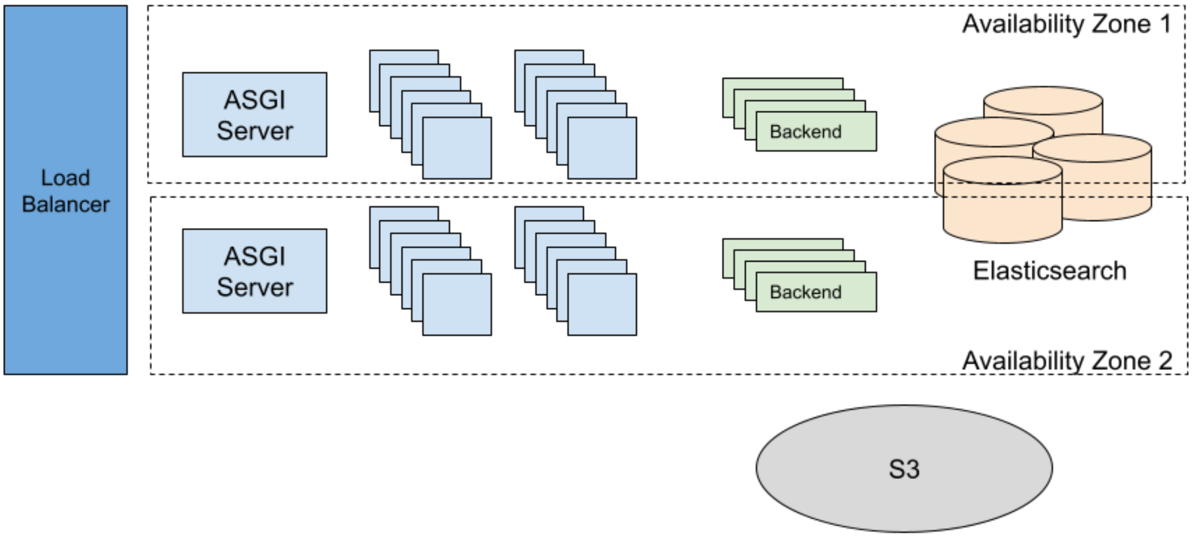

Figure 0.4: Deployment

TODO: move this All ArtifactDB instances share the same features:

- Metadata must follow pre-defined JSON schemas. These schemas correspond to data types. They are converted into "models" used to make this metadata searchable through an efficient indexing engine (Elasticsearch)
- Fine-grained permissions can be defined using a Role Base Access Control (RBAC) pattern
- Data and metadata are organized and grouped as projects, with versioning support (with optional automatic provisioning of project identifiers and versions)
- Authentication is based on JWT tokens
- Each API provides unique Genomics Platform Resource Names, or "GPRNs"(see `artifactdb-identifiers` repo), to easily refer to any given resources within a platform (artifacts, projects, versions, changelog, documentation, etc…)
- Extensible with backend plugins, which can periodically run based on a schedule or based on certain events happening internally within ArtifactDB instances.
- Deployed as high performance, responsive and scalable REST APIs, built on top of Kubernetes, in the cloud.

Configuration files

ArtifactDB configuration can be described in YAML files. The content is organized is different sections, targetting the different components of an ArtifactDB instance, such as Elasticsearch, storages, schemas, etc… A single YAML file can be used, but can also result in a lenghty content. To address this issue, the configuration content can optionally be split across multiple files. These files are loaded and merged sequentially, based on the root-level keys representing *sections* (see below), in alphatical order. If multiple files contains duplicated configuration content, the last merged one will take precedence.

Sections

Throughout the rest of this document, these sections and their sub-fields within them are addressed using the “dot-field” notation. For instance, the section `es` refers to the Elasticsearch configuration, found at the root level of the configuration file. `es.frontend` refers to the sub-section `frontend`, found under the main `es` section, which would be found in the YAML file as:

```
es:
  frontend:
    ...
```

The configuration content is translated into a python structures based on the library aumbry. Each section usually corresponds to one aumbry model, as well as complex sub-sections. The final structure is assembled in a class named `ArtifactDBConfigBase`, mapping sections found in the YAML files.

`!include` constructor

A special YAML constructor allows the usage of the instruction `!include` followed by the path of another YAML file. This enabled the inclusion of other YAML configuration elements, outside of the main configuration files, such as encrypted secrets created during the deployment of the instance, preventing the exposure of password and such, a in clear and unsecure way.

For instance, the file `/app/run/secrets/keycloak/svc-credentials.yaml` contains Keycloakd credentials for a service account. In the context of a Kubernetes deployment, this file comes from a Secret object, injected in the pods filesystem by Kubernetes itself. Without exposing these credentials, the following declaration can be used to inject them at the configuration level:

```
    service_account:
      credentials: !include /app/run/secrets/keycloak/svc-credentials.yaml
```

The resulting python dict structure contains the credentials in clear (so they can actually be used), but to prevent any sensitive information leakage, through printing in logs for instance, all aumbry models derive from a custom `artifactdb.config.utils.PrintableYamlConfig`, which handles redacting configuration data as necessary.

The `!include` constructor tag is a custom one, and trying to load a configuration file with a standard YAML loader will fail because of that. Using the main entry point `artifactdb.config.get_config(...)` function allows to load configuration files using such tags.

Environment

By design, configuration files must include the environment in their filename, following the convention `config-{env}.yml`. Enforcing this prevents from having generic files like `config.yml` being loaded and used, whithout knowing if the content related to a development or production environment. The `{env}` information is usually taken from an environment variable named `ARTIFACTDB_ENV`, set to the proper value when deploying the ArtifactDB instance (eg. `dev` to load `config-dev.yml file). This is the recommended way, but` artifactdb.config.get_config(...)`can take an argument`env‘ as input, for the same result.

Patches

Configuration files are considered read-only, which is actually usually enforced by Kubernetes itself during the deployment (`volumeMount` declared as `readOnly: true`). In some cases, it's useful to modify or enrich the configuration content. This can be achieved by adding files containing JSON patch operations. These files must follow the naming convention `patch-{env}-{section}.yml`. For instance, the patch config file `patch-dev-es.yaml` content:

```yaml
- op: add
  path: /es/frontend/clients/v2
  value:
    alias: myapi-dev-v2
```

instructs the configuration loaded to `add` and frontend client named `v2` in the `es.frontend` section, and this client should use an Elasticsearch alias named `myapi-dev-v2` (more on the configuration itself in the Elasticsearch and indexing part). The resulting patched configuration would look like

```yaml
es:
  frontend:
    clients:
      # content coming from the "read-only" config file
      v1:
        alias: myapi-dev-v1
```

```
    # content coming the patch config file, once the JSON patch operations were applied
    v2:
      alias: myapi-dev-v2
```

During the starting process of the instance, configuration files are loaded, then optional patches are applied in alphetical order, on top the configuration content.

This section was aiming at explaining the configuration options and mecanisms available to “shape” and customize an ArtifactDB instance. The actual content and documentation of each fields are adressed in the corresponding sections later in the documentation.

Schemas

Each instance of an ArtifactDB API corresponds to one specific business use case, usually defined as the type of data the API holds. ArtifactDB heavily relies on JSON schemas to specify the structure of the metadata collected by the instances. Beyond ensuring and enforcing consistent metadata structure, the role of these JSON schemas is fundamental in terms of process as it outlines the frontier between domain-specific activity operated by business users, and the domain-agnostic Data Infrastructure itself.

From a technical perspective, JSON schemas are converted into Elasticsearch DSL models, themselves generating Elasticsearch mappings. These mappings define what to index and how. JSON schemas can optionally be annotated to add extra information for the mapping generation. For example, if a specific field needs to allow partial match searches (as opposed to exact match), it needs to be indexed as a `text` field. Such information cannot be guessed automagically during the conversion step.

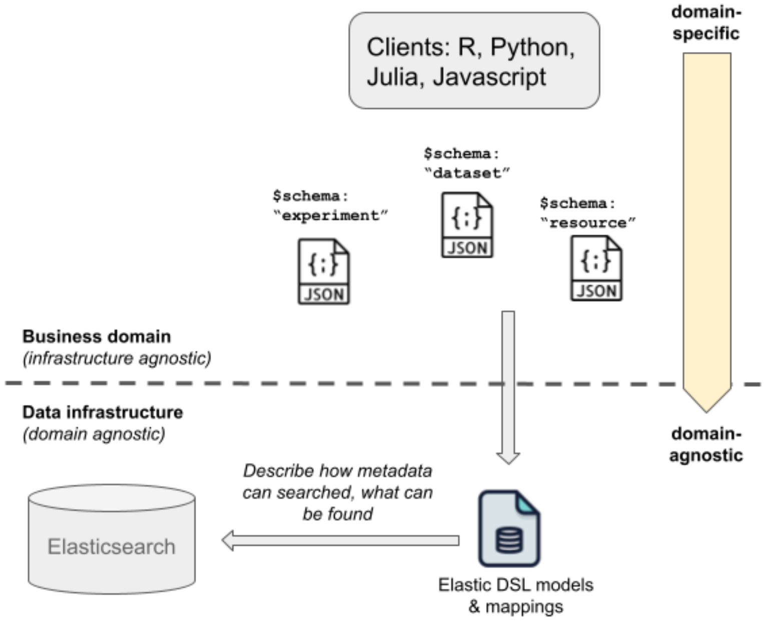

Figure 0.5: JSON schemas as a separation between domain-specific and domain-agnostic activities

Beside these rare exceptions, a clear separation exists between domain-specific (business) and domain-agnostic (infrastructure) activities. This is even more emphasized if we add ArtifactDB clients to the picture. For example, clients handle specific data structures (such as Multi Assay Experiments, MAE, in the R language) and is responsible for collecting data and metadata from the API. These clients are close to the business so they can evolve quickly and efficiently, according to business requirements. When metadata needs to evolve, business users, who are the most knowledgable for specifying metadata, create or update JSON schemas and submit them to the ArtifactDB instance, through this conversion step.

Constraints

In the current implementation, a field `path` must be present in each metadata document, thus at the root ant schema. It represents the underlying data file path, within a project and version directories. This `path` field is mandatory as it is used to build the file identifier, aka. identifiers.

Note: if a metadata file doesn't reference a data file per se, the `path` must still be present. Its value is by convention the path of the JSON metadata file itself.

The content of the schema, ie. metadata structure, can be anything, limited to the depth imposed by Elasticsearch and its configuration, and/or the total numbe of indexed fields. This is usually not an issue, reaching these limits would be an extreme case. Finally, the keys found at the root of the schema must not start with the character `_` (reserved either by the ArtifactDB framework itself, eg. `_extra`, or Elasticsearch, eg. `_id`)

TODO: examples

Storages

Storages play an important role in ArtifactDB, as they are considered the source of truth, for any data, metadata and other administrative metadata (permissions, links, revisions, etc…). Storages are traditionally blob-store living in the cloud, namely AWS S3 for instance. While this is currently the only storage type implement, it's possible to implement more (other blob-store on Azure, GCP, filesystems, or even another ArtifactDB instance).

By design, there is no database, everything is stored on the storage(s). While there is an indexing engine, like Elasticsearch, handling searches over metadata, this can be seen a duplication of metadata stored in an optimized database. Should an index be deleted, the REST API would certainly be impacted, but there would be not data loss, since only the duplicated data is lost.

The other important advantage of this approach is it's easy to understand the content of that storage. After all, an ArtifactDB instance is a receptacle of data and metadata files uploaded by a client, there is no transformation, no modification, no data/metadata deconstruction into an internal, possibly cryptic data model hidden in the system. Transparency is key when it comes to data management, what you upload if what you download. This approach reduces "vendor lock-ins", data can be migrated to another system if necessary, starting from the original content uploaded initially. This also allows data migration to storage in the cloud, using a temporary ArtifactDB instance for that matter, before getting rid of it once done.

Content structure

ArtifactDB hold data and metadata, organized by **projects**, each project can then have multiple **versions**. This is reflected on the folder structure on the storage: one folder per project, one folder per version within a project.

Ex: this storage contains 2 projects, `PRJ00001` with 2 versions (`VER0001` and `VER0002`) and `PRJ00003` with one version (`VER0009`)

```
/PRJ00001/VER0001
/PRJ00001/VER0002
/PRJ00003/VER0009
```

This contrived example illustrates different aspects:

- By convention, projects have a so-called **prefix**, here `PRJ`. One instance can hold more than one prefix (eg. `PRJ` and `test-PRJ` to isolate testing data by naming conversion for instance). There could be padding zero like here, or not (`PRJ1`, `PRJ2`, etc…) depending on the convention put in place in the instance.

- The version is not necessarily an integer. It's actually always treated as a string and can be anything (one example would be a Git commit hash).
- If the instance supports auto-provisioning using one or more sequence(s), these project IDs and versions can be provisioned by the instance, in a transactional way.
- Project IDs and versions may not be sequential, even using auto-provisioning. If an upload fails, the provisioned project ID and/or version is lost, and new ones will be provisioned on the next try.
- Finally, there is currently no way to overcome this structure, there has to be a project folder, and within it, at least one version folder.

Back on the subject of versioning, how to order versions if they're string and can be anything? How to know which version is the latest? During the upload process, ArtifactDB assigns a **revision** along side the version. This revision can be seen as a human readable version, and also provides a way to determine the numerical value of the version, by parsing its value. For instance, assuming an instance provides revision in the format `REV-x` (each instance can be their own way to declare revision format), where `x` is an integer, a version `a1b2c3d4f5g` could have the revision `REV-3` assigned during the upload. The resulting numerical revision would be 3. The API uses this information to determine the order the versions, and the latest one. As any data, metadata and internal metadata files, this revision information stored on the storage, in each version folder, in `..meta/revision.json` internal metadata file.

Ex: example of a `..meta/revision.json` file content

```
{
    "revision": "REV-3",
    "numerical_revision": 3
}
```

Storage types

ArtifactDB is currently heavily oriented towards AWS cloud support, as such, the main storage currently supported in AWS S3. The design itself allows implementation of other storage types (GCP or Azure blob-stores, local filesystem).

Multi-storages

Multiple storages can be declared in an instance. There are many different use cases that benefits from that features, which goes beyond the scope of the document, but to name a few, we can mention data migration, API data versioning, data replication, archiving, etc…

At any point in time though, there is currently only one *active* storage[1]. For instance, when accessing the REST API and fetching data, only one storage can serve that data. The selection of a specific storage can done in different ways:

- A so-called "switch" parameter can be used to specify, depending on the value an HTTP header, which storage should be used (see below the configuration section). Uploading, accessing data can be done by specifying this header (or through proxy rules which can set the header depending on the prefix path for instance).
- Project indexing from a specific storage can be achieved using the admin reserved endpoint `/index/build`, which accepts a `storage` parameter for the storage alias.

The data location is revealed in the internal metadata `_extra.location`, for each JSON document served by the API, indicating the type and information about the storage itself:

```
"_extra":{
    "location":{
        "type":"s3",
        "s3":{
            "bucket":"mybucket-v3-uat"
        }
    },
    ...
}
```

Configuration

The storage configuration can be found under the `storage` section of the configuration file.

- `storage.clients` lists the different storages themselves.
- `storage.switch` declares the rules to switch between storages.

Storages

Each storage must declare:

- a unique `alias` (amongst other storages in the instance)
- a `type` (ex. `s3`)
- and a key named after the type (eg. `s3`). Depending on the type, the value associated to that key changes. For AWS S3 (`type: s3`):

[1] A future implementation might overcome this limitation.

  - `endpoint`: custom S3 endpoint, if not using the default one (or if using another implementation of S3)
  - `bucket`: bucket name
  - `credentials`: dictionary containing the credentials to access the bucket.
  - `presigned_url_expiration`: default TTL in second for presigned-URLs
  - `signature_version`: specific signature version (eg `s3v4`), useful for instance when KMS is involved for the encryption of the bucket itself).
  - `region`: preferred region to access the bucket.
  - `delete_stale_projects_older_than`: clean failed (stale) uploaded project (no properly completed, marked as "to-be-deleted"), after a certain amount of time (eg. "in 2 weeks", meaning all stale projects older than two weeks will be purged). The value must be parsable by the python library dateparser.
  - `bucket_versioning`: ternary value
    * **`null`** (default): the bucket versioning configuration is left intact
    * **`true`**: bucket versioning is enabled
    * **`false`**: bucket versioning is disabled

Switch

When using multiple storages, a switch can be used to specifcy which one to use depending on the request.

- `header`: the HTTP header name based on which the storage selection decision is made.
- `contexts`: map of header value <=> storage alias. When the above header's value matches on of these, the corresponding storage is selected (based on its alias name).

Example

The following example illustrates the usage of multiple storages when upgrading an API. Let's say the API in question needs a significant upgrade, from v2 to v3, which involves breaking changes on the metadata and/or data structure. We still want to keep the v2 data, even serves it as a backward compatibility courtesy. Using two different buckets also make the transition easier, as opposed to mixing v2 and v3 data into the same bucket.

We declare two storages, with an alias `v2` and `v3`, as followed. When accessing the REST API, we can specify a custom HTTP header named `X-MyAPI-Version`. When `v2`, the switch rule sets a storage context with the value of the corresponding alias ("v2" in header mapped to storage alias "v2"). Same for v3.

```yaml
storage:
  clients:
  - alias: v3
    type: s3
    s3:
      bucket: myapi-v3
      credentials: !include /app/run/secrets/s3/credentials.yaml
      signature_version: s3v4
      region: us-west-2
  - alias: v2
    type: s3
    s3:
      bucket: myapi-v2
      credentials: !include /app/run/secrets/s3/credentials.yaml
      signature_version: s3v4
      region: us-west-2
  # v2/v3 switch (matching X-MyAPI-Version header)
  switch:
    header: X-MyAPI-Version
    contexts:  # matching a client alias
      v2: v2
      v3: v3
```

Limits

Limitations, such the max number of files, max total storage size, max total size per file is directly dictated by the storage itself. For AWS S3, a single file cannot be more 5TB (this requires multi-part uploads, easily handled using STS credentials for instance), or 5GB using a single pre-signed upload URLs.

Identifiers

ArtifactDB ID

ArtifactDB ID, also referred as `aid` uniquely identifies an artifact within an ArtifactDB instance. Its syntax is as follows:

```
<project_id>:<path>@<version>
```

When translated to a location in a storage such as S3, it corresponds to the following filename:

```
<project_id>/<version>/<path>
```

Example:

```
PRJ2:folder1/one.file-2.csv@8b60b19b58ef9de4de2e4e8ed8673a4e59491b53
```

where: - `PRJ2` is the project ID - `folder1/one.file-2.csv` is the file path - and `8b60b19b58ef9de4de2e4e8ed8673a4e59491b53` is the version

These different parts can be found in the internal ArtifactDB metadata key `_extra`, in each

Note: ArtifactDB IDs are ensured by design to be unique within an instance, not across instances.

Project ID

The component `project_id` represents a project name, unique per instance. Project name can include letters, usually uppercased by convention, and digits. Specifically, the following characters are not allowed: `/`, a leading `_`, A prefix is also used as a convention, eg. `PRJ`, `ADB`, `DS`, etc… to indicate the type of the project, as a hint. A single ArtifactDB instance can hold multiple project prefixes, see section about sequences for more.

Version

Versions are assembled under a project ID, represent by the component `version` at the end of an ArtifactDB ID, after the `@` character. It can be composed of digits or letters. The same excluded characters listed in `project_id` applies there as well.

A revision name can also be used instead of a version, as a sort of aliasing mechanism. This becomes useful when versions are not human-friendly, such as commit hash. Revision are always generated and assigned by the ArtifactDB instance each time data is uploaded. For instance, the version holding a commit hash `8b60b19b58ef9de4de2e4e8ed8673a4e59491b53` could also be represented by the revision `NUM-3` (assuming this is the third upload), easier to remember. See also the storages section for more on revision and their structure.

Latest revision A special value, `latest` (or `LATEST`), can be used to point to the latest revision of a project or file. `latest` can be used anywhere instead of a version or revision number, and is a useful way to point the latest artifacts. A little bit like the tag "latest" in Docker registries…

Ex: if version `8b60b19b58ef9de4de2e4e8ed8673a4e59491b53`, as revision `NUM-3` is the latest found in project `PRJ2`, the following ArtifactDB ID all represent the same file:

```
PRJ2:folder1/one.file-2.csv@8b60b19b58ef9de4de2e4e8ed8673a4e59491b53
PRJ2:folder1/one.file-2.csv@NUM-3
PRJ2:folder1/one.file-2.csv@latest
PRJ2:folder1/one.file-2.csv@LATEST
PRJ2:folder1/one.file-2.csv@lAtEsT
```

The special version or revision `latest` is useful to always point to the latest data, but not to refer to a specific version. This is important to keep that in mind when running data workflow for instance: if more versions are created, `PRJ2:folder1/one.file-2.csv@latest` would then point to a different version, potentially impacting the reproducibility of the workflow if that one was using a `latest` version tag instead of an explicit version name.

Path

The component `path` is the filename path within the version. It can point to a subdirectory structure, can contain `/` as the traditional delimiter for folders.

Genomics Platform Resource Names (GPRNs)

GPRNs are inspired by Amazon AWS ARNs and uniquely identify a resource within the ArtifactDB Platform. This nomenclature is used by ArtifactDB instances themselves, but also other component of the platform. The name comes from where it originated, the Genomics Plaform, but the "GP" part can easily stand for other more generic names, like "Generic Product", "Global Product", "Great Platform", etc…

A resource is a generic term describing "something" within the platform. It can be an artifact in an ArtifactDB API, it can be an API, an API on specific environment, etc… The format is the following, with some segments being optional or defaulting to specific values or meaning. When omitted, the number of `:` within the GPRN must be kept (this produces things like `::`):

```
gprn:environment:service:placeholder:type-id:resource-id
```

- `gprn`: prefix, mandatory
- *environment*: optionally specify the environment on which the resource can be found. Example: `dev`, `tst`, `prd`, etc… If omitted, the environment is the production.
- *service*: mandatory. The service, application, api, etc… on which the resource can be found. Ex: `myapi`, `yourapi`, etc…
- *placeholder*: is a placeholder, in case another segment is required. (it's `region` in original ARNs)

At the point, the segments allow to uniquely describe a service, on a specific environment. Ex: - `gprn::myapi` means "MyAPI service, production environment" - `gprn:dev:yourapi` means "YourAPI service, development environment" - `gprn::yourapi:europe` means "YourAPI service, production, located in Europe"

Continuing further, we can describe resources within services: - *type-id*: optional if *resource-id* not specified, otherwise required. Type of resource described in *resource-id* - *resource-id*: optional if *type-id* not specficied, otherwise required. ID of type *type-id* within the service.

Ex: - `gprn::myapi::artifact:PRJ2:result.html@PUBLISHED-3` means the Artifact ID `PRJ2:result.html@PUBSLIHED-3` in MyAPI, production. - `gprn::myapi::project:PRJ2` means project PRJ2 within MyAPI, production - `gprn::myapi::project:PRJ2@NUM-3` means project `PRJ2`, version `NUM-3`, within MyAPI, production - `gprn::myapi::doc` means the documentation for MyAPI API.

GPRNs can found in metadata returned by ArtifactDB APIs, which also provide dedicated endpoints to help manipulating them.

Authentication

ArtifactDB REST API provides authentication based on JWT (JSON Web Token) and the OpenID Connect implementation. A JWT is an encoded and signed JSON document, containing header and claims. The header contains references to the public key that can be used to verify the signature, while the claims contains the actual payload: the name of the user, the client ID, the roles, etc…

JWT based authentication

Upon reception of an HTTP request from a client, a FastAPI "authentication" middleware intercepts that request to extract the authentication information. Specifically, it expects that such information, if present, must be found in the header `Authorization`, with a value looking like `Bearer token_string`[2]. The auth middleware proceeds by validating the token string, by downloading (then caching) the public key ID `kid` found in the JWT header.

The configuration regarding authentication allows to declare different so-called "well-known" URLs, which according to the OpenID standard, must provide URL to the content of the public key as well as its `kid`. To allow flexibility in the JWT sources, the configuration allows to declare:

- one primary well-known URL: this is the main one, used to authenticate users through Swagger (section `auth.oidc.well_known.primary`).
- several secondary well-known URLs: alternately, these URLs are used to also verify the signature (section `auth.oidc.well_known.secondary`).

These public keys don't change often and constantly fetching their value would be a waste of resources. More, if the OpenID provider is not available, the signature verification would fail and the API would not be able to perform the request and serve the client. To avoid this (dramatic) situation, an agressive caching mecanism is put in place: if the well-known URLs or the public key URLs are not available in a timely manner, a previously never expiring cache is used as a source. Upon each (re)start of the API, the cache is updated if possible.

To facilitate integration with other external systems, a list of clients (section `auth.clients.known`) can optionally be provided. The client ID found in the JWT token is matched against the instance's main client ID (`auth.oidc.client_id`) or one of these. If there's no match, the middleware rejects the request with an explicit `HTTP 400` error, signaling the token provided by the client is invalid (with a self-explanatory message). Depending on the token format, the client ID is taken in from the field `azp`, or `client_id`. If both `azp` and `client_id` are found in the claims, `client_id` has precedence over `azp`.

[2] https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-bearer-19#page-5

Once the signature has been verified, the middleware also verifies that the token is still valid, by checking the claims field `exp`. This field contains a timestamp in the Epoch format and assumed to be in UTC. Token caching is possible (though rarely used) by enabling the parameter `auth.oidc.cache_tokens`. When true, the field `jti` (JWT ID, a unique token ID value) is used for that purpose, as well as the `exp` value (minus a few seconds to allow some slack in the expired/not-expired decision) for the cache TTL value. If the same token is used and was cached by the API, the verification process (signature, expiration) is skipped, to allow faster request processing.

Passing these checks, the middleware inspects the potential roles found in the claims, in order to know if the user is an admin or not. An instance of an `artifactdb.rest.auth.AuthenticatedUser` (or `artifactdb.rest.auth.AdminUser`) is created, and injected in a context variable for the time of the request. This "user context" variable is used when querying Elasticsearch to inject authorization information (permissions), avoiding to pass this user across all function and method calls. By design, if no user is found in this context variable, the Elasticsearch query fails (this, to prevent any accidental data access, ie. no user could be lead to no permission-based filtering in the query). Once the request was processed (a response was prepared, ready to return), the middleware reads the user context variable, checks that the value is the same than the one it previously set, before resetting the context itself. This extra check at the end ensures there was no accidental manipulation of the context itself during the request, as well as ensure the next request starts from a fresh, empty context.

This user object contains information such as her `unixID` (taken from `preferred_username`), the resource she's trying to access (the path), the roles if any, the raw and parsed JWT token. Optionally, the middleware can enrich distribution lists and Active Directory groups membership, if such information providers are declared in the configuration.

Finally, if no JWT token is found is the HTTP header, the request is considered anonymous. Following the user context variable requirement, an `artifactdb.rest.auth.AnonymousUser` instance is created and injected in the context. Without this, all queries would fail. The query engine (`artifactdb.db.elastic.manager.ElasticsearchManager`) inspects the user type (anonymous, authenticated, admin) to adjust the permission-based filtering rule and return results accordingly (eg. only publicly readble projects for anonymous users).

"IKYS" API key based authentication

For very specific use cases, another authentication method is available. IKYS ("I Know Your Secret") API key can be used when a component of an ArtifactDB deployment needs to gain access to the REST API as "admin". The authentication method is based on the fact that this component (eg. a pod living in the same namespace as the instance) can have access to the same secrets as the instance itself. That component is able to know one of the secrets of the instance, secrets which are never exposed outside of the Kubernetes deployment itself.

Enabling this authentication requires to pass a custom HTTP header named `X-API-IKYS-Key`. Its value is an base64 encoded JSON document about the secret. Ex:

```
{
    "type": "ikys",
    "hash_function": "sha256",
    "hashed_secret": "abcazer145...",
    "secret_path": "./path/to/secret/on/frontend/pod/side"
}
```

The secret path is tells the REST API where the secret the other component is located and knows about. The file hashed using the passed hashed function, the hexdigest put in the JSON document.

Again, the authentication method fits very special use cases, implying access to the internal instance deployment. Such use case is a Kubernetes operator, deployed along side the instance, which needs full access to the REST API to control and operate on it. It is not meant to be made for external users or services, it not a “classic” API key.

Backend components

The ArtifactDB server-side components are known as the "ArtifactDB backend". This naming comes from the facts multiple clients can talk to the server REST API, enforcing client vs. backend distinction. That said, the server components themselves can be decomposed into "frontend" and "backend" elements:

- *frontend*: everything relating the REST API, public facing. That's the server-side component the clients are talking to.
- *backend*: evertyhing that is happening behind and beyond the frontend, most of the time, as asynchronous tasks.

This section describes this backend component specifically.

Backend Manager

The backend component is organized around a "manager", gathering multiple sub-components by composition. Some of these sub-components are required, some have dependencies between each others, and some are optionals but add specific features (eg. external integration, instance-specific features, etc…). Creating a backend manager instance requires some modularity to fit the different use cases and to allow extendability. This flexibility is implemented by declaring the component classes at the manager's class level, the manager itself, during its instanciation, builds the sub-components and integrates the

```
import artifactdb.backend.managers.base
import artifactdb.backend.sequences
import mycustom.builder.notifications

class MyBackendManager(artifactdb.backend.managers.base.
    BackendManagerBase):

     COMPONENTS = [
         {"module": artifactdb.backend.sequences, "required": True},
         {"module": mycustom.builder.notifications, "required": False}
     ]
```

This example declares a `MyBackendManager` class with two components, one required, one not required. When a component is required, if its creation fails, the whole backend manager instanciation is declared failed. Failures for non-required components are logged but ignored. During initialization, the manager will inspect the modules declared in the list `COMPONENTS`, looking for classes inheriting from either:

- `artifactdb.backend.components.BackendComponent`: the whole component logic is fully implemented in the component class itself, including the constructor.
- or `artifactdb.backend.components.WrappedBackendComponent`: this class acts as a wrapper over an existing class living outside of the the context of backend components. The constructor and wrapping logic is taken care of the `WrappedBackendComponent`, only the method `wrapped()` requires to be implemented in the sub-class. For instance, `ElasticSearchManager` is an important element of an ArtifactDB API responsible for querying, indexing data. It lives on its own, is used by the REST API. A wrapped component is typically used in this case to easily convert this class into fully functional component.

Either ways, the whole configuration and the backend manager instance itsefl are provided and made available to the sub-component during its instantiation, it's the responsability of the sub-component to select which part(s) of the configuration should be used. The dependencies that may exists between these sub-components are adressed with the order by which they appear in the `COMPONENTS` list.

Each time a new sub-component is added, one or more *feature* can be registered as well. In the previous example, such *features* could be `"auto-provisioning"` (sequence number, that is, project IDs), and `"notifications"`. These features are later exposed by the REST API to inform users and clients of the instance capabilities currently available.

During the backend and its components initialization, several "hooks" can be implemented to enrich this step, at different stages. Indeded, it is not rare that a component needs extra initialization steps later, after its own creation. The following hooks are called in `artifactdb.backend.app.get_app()`, which is the main entry point in the backend to initialize the backend manager and link it to the main Celery application. The following hooks are called for each successfully registered components:

- `component_init()`: this method is called just after the component instance has been created, while the components are discovered and registered.
- `post_manager_init()`: this method is called just after the backend manager instance has been created. Components are fully registered at that point, but the backend tasks (Celery tasks) are not registered yet. This hook is useful when a component needs to prepare some work before these tasks, for instance pulling their code from Git repositories (that's what the `PluginsComponent` does)
- `post_tasks_init()`: once tasks are registered, this method is called. This hookd is useful for instance to obtain information about these tasks (this is what the `TasksComponent` does).
- `post_final_init()`: finally, before returning the final Celery application and its linked backend manager, this method is called for "last-call init".

Transient

Sequences and auto-provisioning

In the ArtifactDB world, data and metadata files are organized in projects, then versions. This means project must have unique IDs across them, and versions within a single project must also be unique. There are different options achieving this requirement: external provisioning vs. auto-provisioning.

External provisioning

The uniqueness of projects ID and versions are left to the client interfacing wit the ArtifactDB instance. This is useful for instance when these identifiers are stored and managed in an external system. The instance will not provision any IDs, nor will it ensure an existing project is not being overwritten. It just trusts and follow the directions communicated by the client.

Auto-provisioning

When this component is enabled, a `artifactdb.backend.components.sequences.SequenceManager` is created and attached to the backend manager. This manager holds one or more `artifactdb.backend.components.sequences.SequenceClient` responsible to provision project identifiers, and for each, unique versions, in a transactional way. The underlying implementation involves a SQL database and a sequence (if Postgres), so the name. The library `sqlalchemy` is used for the SQL database operations. A sequence for a project ID is usually defined as a project prefix (eg. `PRJ`, `DS`, etc…), while the version is based on on incremented integer, per project.

Synchronizing

Following one of the core principle of ArtifactDB, "the storage (like s3) is the source of truth", the whole SQL database content can be derived from the storage content. Projects and their versions are listed and used to populate the initial content. During that "sync" operation, the table containing the actual sequence information is locked, to prevent any concurrent writing (a new project being created while syncing the sequences). Any request for provisioning, ie. any upload requests, will hang until the sequence content is restored from the storage. This operation can take a long time, and the instance should be put in a maintenance mode to inform users new uploads should be postponed.

Each `SequenceClient` is associated, per configuration (see below), to one sequence content. The synchronization process is performed from the sequence client, which means if multiple sequences are declared (eg. multiple prefixes), each can be synchornized independently from the others.

TODO: link to "Maintenance mode"

Pools

Pools declare intervals to contrain project identifiers values. They come in two different flavors:

- *provisioned pool*: declares an inclusive interval within which a project ID will be provisioned. There can be only one active provisioned pool, if a new provisioned pool is created, the previous one will set to `inactive`.
- *restricted pool*: declares an inclusive interval within which a project ID is *not* allowed to be provisioned. There can be multiple restricted pools at a time.

Restricted pools can overlap a provisioned pool, in which case the overlapped IDs are removed from the provisioning process (in other words, restricted pools have precedence over provisioned pools).

When starting for the first time, the ArtifactDB instance will create a provisioned pool ranging from 1 up to `max_sequence_id`, with 999'999'999 being the default, if `auto_create_pool` is true (default). See the section below for more on sequence configuration).

TODO: image pools, provisioned active/inactive, restricted, overlapping

Interfaces

API endooint The sequence information (project ID and version) are used while uploading artifacts, see section upload for more. Other endpoints can be used to collect information about the different sequences state. These require by default the role `admin`.

- `GET /sequences` returns exhautive information for all sequence clients, including the project prefix, the provisioned and restricted pools, the last provisioned ID, etc...
- `GET /sequences/{project_id}/current_version` can be used to obtain the last version assigned to a give project ID.

Configuration Using a sequence requires to declare a list of configuration elements. Each element of that list will produce a `SequenceClient`. The `uri`, `db_user`, `db_password` and `schema_name` parameters specify the database access are roughly passed to `sqlalchemy.create_engine(...)` function. The remaining parameters drive the sequence behavior, specifically the project ID naming.

The whole sequence configuration is declared under the root key `sequence`, containing a list of configuration elements, one element per sequence client:

```
sequence:
  # sequence client 1
  - uri: ...
```

```
    db_user: ...
    ...
  # sequence client 2
  - uri: ...
    db_user: ...
```

The sequence client can configured with the following parameter:

Parameter	Description
`uri`	sqlalchemy compliant database URI
`db_user`	username used for the database connection
`db_password`	password for that user (using "!include /path/to/secret" tag is recommended (TODO: link to config)
`schema_name`	schema name (postgres), or database name (mysql)
`project_prefix`	short prefix for all project IDs, eg. PROJ, DS, etc…
`project_format`	project format rule, as f-string, ex: 'f"{project_prefix}{seq_id:09}"'. `seq_id` is passed by the sequence client itself and corresponds to the incremented integer returned by the SQL sequence itself. The whole f-string is then `eval`'d to obtain the final result. The version format is left to the sequence client implementation (default is an incremented integer).
`max_sequence_id`	upper limit for the project ID, defaulting to 999999999
`version_first`	specifies the first version to provision for a new project, defaulting to "1"
`auto_create_pool`	upon fresh start, auto-create provisioned pool from [1,max_sequence_id]
`default`	if no sequence prefix is specified, instruct the `SequenceManager` to use this sequence by default

Parameter	Description
`debug`	activate SQL debug statements for troubleshooting

Also see `artifact.config.sequences` module and `SequenceConfig` class for more.

Administration Sequences can be manipulated from an *admin* shell to perform maintenance operations. Because these operations are rare and critical, there are not available through endpoints. Let's see some examples.

TODO: link to "admin shell"

Manipulating provisioned and restricted pool

```
# list sequence clients
> mgr.sequence_manager.clients
{'RDB': <SequenceClient (schema='rdbseq_dev_rdb', prefix='RDB', default
    =True)>,
 'test-RDB': <SequenceClient (schema='rdbseq_dev_test_rdb', prefix='
    test-RDB', default=False)>}
# fetch the sequence client we're interested in
> seq_client = mgr.sequence_manager.clients["RDB"]
# list existing pools
> seq_client.list_provisioned_pools()
[{'pool_id': 1,
  'pool_type': 'PROVISIONED',
  'pool_status': 'ACTIVE',
  'lower_limit': 2,
  'upper_limit': 999999999,
  'created_at': datetime.datetime(2022, 6, 21, 19, 47, 5, 898703,
      tzinfo=psycopg2.tz.FixedOffsetTimezone(offset=0, name=None))}]
> seq_client.list_restricted_pools()
[]
# check current project ID
> seq_client.current_id()
"RDB000000003"
# obtain a new one
> seq_client.next_id()
'RDB000000004'
# the next one would 5, etc... Let's restrict this and forbids
    provisioning from [5,10]
> seq_client.create_restricted_pool(5,10)
> seq_client.list_restricted_pools()
[{'pool_id': 2,
  'pool_type': 'RESTRICTED',
  'pool_status': 'ACTIVE',
  'lower_limit': 5,
```

```
  'upper_limit': 10,
  'created_at': datetime.datetime(2022, 9, 27, 17, 31, 55, 92508,
     tzinfo=psycopg2.tz.FixedOffsetTimezone(offset=0, name=None))}]
# the restricted pool overlaps the provisioned one, next ID is...
> seq_client.next_id()
'RDB000000011'
# indeed, not 5 but 11! Let's create a new provisioned pool
> seq_client.create_provisioned_pool(100, 200)
# list all provisioned, regardless of `pool_status`
 seq_client.list_provisioned_pools(pool_status=None)
Out[16]:
[{'pool_id': 3,
  'pool_type': 'PROVISIONED',
  'pool_status': 'ACTIVE',
  'lower_limit': 100,
  'upper_limit': 200,
  'created_at': datetime.datetime(2022, 9, 27, 17, 33, 38, 519968,
     tzinfo=psycopg2.tz.FixedOffsetTimezone(offset=0, name=None))},
 {'pool_id': 1,
  'pool_type': 'PROVISIONED',
  'pool_status': 'INACTIVE',
  'lower_limit': 2,
  'upper_limit': 999999999,
  'created_at': datetime.datetime(2022, 6, 21, 19, 47, 5, 898703,
     tzinfo=psycopg2.tz.FixedOffsetTimezone(offset=0, name=None))}]
# the old one is now inactive, the new one active. Next ID is...
> seq_client.next_id()
'RDB000000100'
```

Synchronizing sequence content If there was an issue, such as loosing the SQL database, wrong manipulation of pools, or when an instance content was taken from another instance (eg. `aws s3 sync ...` between buckets), potential resulting in a desync with the sequence content, we can ask for a given sequence client to restore its content from the storage.

```
# we'll sync only the `RDB` sequence for example
> seq_client = mgr.sequence_manager.clients["RDB"]
# dry-run to first have a look at the situation
> seq_client.sync(dryrun=True)
DEBUG:root:Active storage: <S3Client bucket=some-bucket>
DEBUG:root:Listing all projects in s3 starting with RDB
...
DEBUG:root:Listing artifacts/versions currently in sequence
...
Do you want to initialize the sequence 'RDB' with the following content
   ? [y/N]
...
(some information about pools the sequence wants to create to match the
    storage content)
...

```

```
y
Let's go...
DEBUG:root:Skipping creation of provision pool
INFO:root:Dry-run mode, rolling back
# remove dry_run to proceed for real
> seq_client.sync()
...
```

Events

TODO: events published to Hermes, data lifecycle

Links

TODO: describe the special links.json documents

Redirections

TODO: explain the special document with redirection schema

TODO: backend plugins

Metadata post-processing

TODO: describe the special jsondiff.json documents to enrich metadata on the backend side

Patterns and Anti-Patterns

ArtifactDB relies on AWS S3 (storage) and Elasticsearch (indexing engine). The backend runs asynchronous tasks, even if triggered from the REST API (eg. reindexing a project). Considering these three core aspects, here are some considerations whether to choose to use ArtifactDB (or not):

- AWS S3 is cheap, highly durable and available. The size limit for a given file is 5TB. S3 is slower than a local disk, or even a local network drive in most cases, though. If a use case involves heavy reading, a cache layer might be necessary.
- The REST API itself doesn't handle the downloads or uploads of data files, but rather delegates that to S3 itself, through the usage of pre-signed URLs, or by providing STS credentials, to benefit from accessing S3 with standard AWS SDK. The bandwidth is thus delegated to AWS S3 itself.
- Elasticsearch is a distributed indexing engine. It thrives at searching and fetching documents by keyword or full text, but can be limiting when it comes to analytical queries compared to a RDBMS. Aggregations are possible, ArtifactDB exposes an endpoint for that purpose, but more advanced queries may required an additional storage system. Though ArtifactDB uses distributed lock and SQL sequences to provision project identifiers and versions, Elasticsearch itself is not transactional, it's an eventual consistency system (reading what's just been written may not be the same, but eventually will). The size of a document, that is the size of a given metadata file, cannot exceed 10MB on AWS. The number of documents can be in the order of tens of billions, as Elasticsearch can scale out easily as more compute, storage and money is thrown at it…
- Indexing metadata is asynchronous, which allows the REST API to stay responsive as it delegates the task to the backend. This approach is usually fine but in some context, again usually transactional, where a response is required as soon as the request was made, for instance as a confirmation or validation, asynchronicity might be a problem.

Deployment

TODO: describe standard Helm chart content/org frontend+backend+beat+admin w/ redis/rabbitmq clusters

Administration pod

Most operations needed to interact with an ArtifactDB API can be performed using the REST API. Some critical, unusual, unexpected operations may require access to an internal administration pod. This pod holds all the code currently running the instance. Using a special script named `tools/admin.py`, a backend manager instance is created, allowing to explore the instance by directly using ArtifactDB SDK python framework, within the instance's context.

The `tools/admin.py` script allows a wide range of usages, as it allows to interact and even program the instance, accessing all its internals. Some useful commands are sometimes described in this documentation. Please also refer to the admin terminal section for more.

ˆ The `tools/admin.py` is available in backend, frontend, even if the admin pod is not enabled during deployment. A admin user with access the Kubernetes namespace the instance is installed in can load that script and achieve the same result as going through that admin pod and terminal. The main advantage of using an admin pod is to securely expose that pod within a web browser, without having to onboard that user on the cluster.

Web TTY (wetty) terminal

This administration pod is available when its deployment is enabled in the Helm chart during the instance deployment. The pod's Docker image is the same as the frontend and backend pods. In addition, a wetty server is running[3] and exposed as a service and an ingress route with the path suffix `/shell`. Admin users can access the pod from a browser and "land" in the pod. They are authenticated based on the Linux users created on the pod. A default list of admin users can be created as an Kubernetes secret, named `admins-credentials` with a data filename `users.txt` in the following format:

```
user1:passwd1
user2:passwd2
```

After the creation of the users, the file content is passed to `chpasswd` to assign password. Passwords are listed in clear, ie. not encrypted (which is fine since the Kubernetes secret is never exposed). If the content of that file is empty, there's no admin users created at all. Any creation of users/passwd manually done from within the pod itself (assuming the person doing this has access to the cluster) will not survice a pod restart, the `admins-credentials` secret serves as a "database" for that purpose.

[3] Because not a single NodeJS application was designed to be easily installed, without major struggle, another approach is used here to deploy that `wetty` application. An initContainer based on the official `wetty` image is used to copy required executables, libraries and source code, into a shared Kubernetes volume. That volume is then made available in the admin container itself. `wetty` is then started, within the python environment provided by the API image.

Security and best practices

While the admin pod and terminal is not publicly accessible, the authentication is based on Linux login/password combination. This should give enough security provided strong passwords are used. Yet, this admin terminal is probably not useful most of time but only in some specific use cases and administration requests. It should *not* be made available unless required. The Kubernetes deployment object can be used to scale it, with `0` to disable the terminal (which would be the nominal state most of the time) and `1` to spin up an admin pod. This, again, requires access to the cluster.

Another alternative, if the instance runs along with an Olympus Maintainer Operator[4], is to create a maintenance request on the API side, asking to scale the admin deployment to 0 or 1. The `PUT /maintenance/requests` endpoint can be used, with the following payload:

```
{
    "name": "scale-deployment",
    "args": ["admin",0]
}
```

where `admin` is the ArtifactDB component to scale, `admin` is this case, and `0` is the number of pod we want, here scaling down to 0, disabling the terminal. `1` can be used to enable the terminal again. Values greater than 1 must be avoided, there can be only one terminal at once.

The operator would then take care of that request automatically. Creating maintenance request requires the role `admin` to be present in the JWT token. Since tokens are temporary and are different each time they're generated, the security is improved when compared to leaving a pod running with static Linux user/passwd.

TODO: link to deploy TODO: link to admin shell usage

[4] The Olympus Maintainer Operator is a Kubernetes operator responsible for keeping the instance healthy and perform maintenance operation. You can read more of this operator here

Conclusion

Wow that was great.