ENCODE Utils Documentation

Release 2.8.0

Nathaniel Watson

Jan 20, 2022

Contents

Installation and configuration instructions are provided on the project's GitHub wiki.

CHAPTER 1

Examples

1.1 Examples

1.1.1 Imports

```
import encode_utils as eu
from encode_utils.connection import Connection
```

1.1.2 Connecting to the production and development Portals

You'll need to instantiate the `Connection` class, passing in a value for the `dcc_mode` argument.

```
#prod portal:
conn = Connection("prod")

#dev portal
conn = Connection("dev")
```

You can provide a custom host name as well, such as a demo host, granted that you have access to it:

```
conn = Connection("demo.encodedcc.org")
```

Dry Runs

The second argument, `dry_run`, can be set to `True`, which allows you to test things out without worrying about any ENCODE Portal modifications being made:

```
conn = Connection("dev",True)
```

The logging will indicate that you are in dry-run mode. Once you are happy with the simulation, you can switch to live-run mode - the default when `dry_run` isn't set:

```
conn.set_live_run()
```

And you can even switch back to dry-run mode:

```
conn.set_dry_run()
```

1.1.3 Log Files

Each time you create a new `Connection` object, either directly as show above, or indirectly through use of the packaged scripts, a log directory by the name of *EU_LOGS* will be created in the calling directory. Three log files are created:

1. A debug log file that contains all of STDOUT.
2. An error log file that contains only terse error messages. This is your first stop for checking to see if any errors occurred. Anything that is written to this file is also written to STDOUT, hence the debug log as well.
3. A POST log file, which only logs new records that are successfully added to the ENCODE Portal. Everything written to this file is also written to STDOUT, hence the debug log as well.

These log files are specific to the host that you connect to. Each host will have a different trio of logs, with the host name included in the log file names.

1.1.4 GET Request

Retrieve the JSON serialization for the Experiment record with accession ENCSR161EAA:

```
conn.get("ENCSR161EAA")
```

1.1.5 Search

Search for ChIP-seq assays performed on primary cells from blood:

```
query = {
    "assay_title": "ChIP-seq",
    "biosample_type": "primary cell",
    "organ_slims": "blood",
    "type": "Experiment"
}

conn.search(query)
```

The query will be URL encoded for you. If you want to use the search functionality programmatically, you should first test your search interactively on the Portal. The result will be an array of record results, where each result is given in its JSON representation.

1.1.6 PATCH Request

Add a new alias to the GeneticModification record ENCGM063ASY. Create a payload (*dict*) that indicates the record to PATCH and the new alias. The record to PATCH must be indicated by using the non-schematic key *Connection.ENCID_KEY*, or *self.ENCID_KEY* from the perspective of a *Connection* instance, which will be removed from the payload prior to submission:

```
payload = {
    conn.ENCID_KEY: "ENCGM063ASY",
    "aliases": ["new-alias"]
  }

conn.patch(payload)
```

1.1.7 File Upload

Given the File record ENCFF852WVP, say that we need to upload the corresponding FASTQ file to AWS (i.e. maybe the former attempt failed at the time of creating the File record). Here's how to do it:

```
conn.upload_file(file_id="ENCFF852WVP")
```

This will only work if the File record has the *submitted_file_name* property set, which is interpreted as the local path to the file to submit. You can also explicitly set the path to the file to upload:

```
conn.upload_file(file_id="ENCFF852WVP",file_path="/path/to/myfile")
```

1.1.8 POST Request

Let's create a new File record on the Portal that represents a FASTQ file, and automatically upload the file to AWS once that is done:

```
payload = {
    "aliases": ["michael-snyder:SCGPM_SReq-1103_HG7CL_L3_GGCTAC_R1.fastq.gz"],
    "dataset": "ENCSR161EAA",
    "file_format": "fastq",
    "flowcell_details": {
      "barcode": "GGCTAC",
      "flowcell": "HG7CL",
      "lane": "3",
      "machine": "COOOPER"
    },
    "output": "reads",
    "paired_end": "1",
    "platform": "encode:HiSeq4000",
    "read_length": 101,
    "replicate": "michael-snyder:GM12878_eGFP-ZBTB11_CRISPR_ChIP_input_R1",
    "submitted_file_name": "/path/to/SCGPM_SReq-1103_HG7CL_L3_GGCTAC_R1.fastq.gz"
}
```

Notice that we didn't specify the required *award* and *lab* properties (required by the ENCODE profiles). When not specified, the defaults will be taken from the environment variables *DCC_AWARD* and *DCC_LAB* when present. Otherwise, you will get an error when trying to submit. Before we can POST this though, we need to indicate the profile of the record-to-be.

Specifying the profile key

We are almost ready to hand this payload over to the *post()* method, however, we need to first indicate the profile to POST to. To do this, add a special key to your payload that is stored in the constant *Connection.PROFILE_KEY*. The *post()* method depends on this key as the way of indicating which profile to create a new record under. There are a few ways in which you can specify the profile, but the recommended way is to use the stripped-down profile ID. If you look

at the JSON schema for the File profile at https://www.encodeproject.org/profiles/file.json, you'll find that the value of it's *id* property is *"/profiles/file.json"*. The stripped-down value that you should use is *file*. Another way to say it is to use the barebones profile name that you put in the URL to get to it. See the documentation in the *profile.Profile* class for further details on how this works.

Without futher ado, let's now add the profile specification to the payload and POST it:

```
payload[Connection.PROFILE_KEY] = "file"
conn.post(payload)
```

The logging to STDOUT and your log files will indicate the progress of your request, including the upload of your FASTQ file to AWS.

1.1.9 Removing properties from a record

This feature is implemented via the PUT HTTP method, which works by replacing the existing record on the Portal with a new representation. You just need to specify a list of property names to be removed. A GET on the record is first made with the query parameter `frame=edit`, and the properties that you indicate for removal are popped out of the returned JSON representation of the record. This updated JSON representation is then sent to the Portal via a PUT operation.

For example, say you have a biosample record and you want to remove the *pooled_from* property. This property stores a list of other biosample records. You can't just empty out the list interactively in the Portal, or programmatically via a PATCH operation since this property, when present, can't be empty. This is where the PUT HTTP method comes in handy. Let's look at an example:

```
conn = Connection("dev")
conn.remove_props(rec_id="ENCBS133ZSU",props=["pooled_from"])
```

It's as simple as that. It should be mentioned that the `remove_props()` method will do some validation of its own to ensure that you aren't trying to delete something that you really shouldn't delete, such as properties that are:

1. required,
2. read-only, and
3. non-submittable.

as indicated in the profile (JSON schema) of the record of interest. The Portal would most likely reject or silently ignore any attempt to remove such properties, nonetheless, to be a good citizen, this client performs these checks regardless for good measure.

It should also be noted that some properties simply can't be deleted. For example, any attempt to delete the *aliases* property will only empty out its list.

1.2 Transferring files to GCP

Encapsulates working with the Google Storage Transfer Service (STS) to transfer files to GCP from either list of public URLs, or from a list of AWS S3 URIs. The latter requires that the user has AWS keys with permissions granted to the source S3 bucket and objects.

The Google Storage Transfer API documentation for Python is available here for more details. This package exports a few ways for interfacing with this logic.

1. A module named *encode_utils.transfer_to_gcp* that defines the *Transfer* class, which provides the ability to transfer files from an S3 bucket (even non-ENCODE S3 buckets) to a GCP bucket. It does so by creating what the STS calls a transferJob. A transferJob can also be created by passing in a URL list, which is a public file accessible through HTTP/S; GCS details at https://cloud.google.com/storage-transfer/docs/create-url-list.
2. The *encode_utils.connection.Connection* class has a method named *encode_utils.connection.Connection.gcp_transfer_from_aws()* that uses the above module and is specific towards working with ENCODE files. Note: this requires AWS keys. If you simply want to copy released ENCODE files, you should When using this method, you don't need to specify the S3 bucket or full path of an S3 object to transfer. All you need to do is specify an ENCODE file identifier, such as an accession or alias, and it will figure out the bucket and path to the file in the bucket for you.
3. The method `encode_utils.connection.Connection.gcp_transfer_from_urllist()`, which doesn't require AWS keys since it only works with released ENCODE files. This method just creates the file that can be used directly as input to the Google Storage Transfer Service in the Google Console, or programatically to the method *encode_utils.transfer_to_gcp.Transfer.from_urllist()*.
4. The script *eu_s3_to_gcp.py* can be used, which calls the aforementioned method *gcp_transfer_from_aws* method, to transfer one or more ENCODE files to GCP. Requires AWS keys.

1.2.1 The Transfer class

Create a one-off transferJob that executes immediately

This example is not ENCODE specific.

```
import encode_utils.transfer_to_gcp as gcp

transfer = gcp.Transfer(gcp_project="sigma-night-206802")
```

The *encode_utils.transfer_to_gcp.Transfer.from_s3()* method is used to create what the STS calls a transferJob. A transferJob either runs once (a one-off job) or is scheduled to run repeatedly, depending on how the job schedule is specified. However, the *from_s3* method shown below only schedules one-off jobs at present:

```
transfer_job = transfer.from_s3(s3_bucket="pulsar-encode-assets", s3_paths=["reads.
↪fastq.gz"], gcp_bucket="nathankw1", description="test")

# At this point you can log into the GCP Consle for a visual look at your transferJob.
transfer_job_id = transfer_job["name"]
print(transfer_job_id) # Looks sth. like "transferJobs/10467364435665373026".

# Get the status of the execution of a transferJob. An execution of a transferJob is␣
↪called
# a transferOperation in the STS lingo:
status = gcp.get_transfer_status(transfer_job_id)
print(status) # i.e. "SUCCESS". Other possabilities: IN_PROGRESS, PAUSED, FAILED,␣
↪ABORTED.

# Get details of the actual transferOperation.
# The "details" variable below is a list of transferOperations. Since we created a␣
↪one-off job,
# there will only be a single transferOperation.
details = transfer.get_transfers_from_job(transfer_job_id)[0]
>>> print(json.dumps(d, indent=4))
#  {
#      "name": "transferOperations/transferJobs-10467364435665373026-1532468269728367
↪",
```

(continues on next page)

(continued from previous page)

```
#       "metadata": {
#           "@type": "type.googleapis.com/google.storagetransfer.v1.TransferOperation",
#           "name": "transferOperations/transferJobs-10467364435665373026-
↪1532468269728367",
#           "projectId": "sigma-night-206802",
#           "transferSpec": {
#               "awsS3DataSource": {
#                   "bucketName": "pulsar-encode-assets"
#               },
#               "gcsDataSink": {
#                   "bucketName": "nathankw1"
#               },
#               "objectConditions": {
#                   "includePrefixes": [
#                       "cat.png"
#                   ]
#               }
#           },
#           "startTime": "2018-07-24T21:37:49.745522946Z",
#           "endTime": "2018-07-24T21:38:10.477273750Z",
#           "status": "SUCCESS",
#           "counters": {
#               "objectsFoundFromSource": "1",
#               "bytesFoundFromSource": "80376",
#               "objectsCopiedToSink": "1",
#               "bytesCopiedToSink": "80376"
#           },
#           "transferJobName": "transferJobs/10467364435665373026"
#       },
#       "done": true,
#       "response": {
#           "@type": "type.googleapis.com/google.protobuf.Empty"
#       }
#   }
```

1.2.2 The *gcp_transfer_from_aws()* method of the *encode_utils.connection.Connection* class

Requires that the user has AWS key permissions on the ENCODE buckets and file objects.

```
import encode_utils.connection as euc
conn = euc.Connection("prod")
# In production mode, the S3 source bucket is set to encode-files. In any other mode,␣
↪the
# bucket is set to encoded-files-dev.

transfer_job = conn.gcp_transfer_from_aws(
    file_ids=["ENCFF270SAL", "ENCFF861EEE"],
    gcp_bucket="nathankw1",
    gcp_project="sigma-night-206802",
    description="test")
```

1.2.3 Copying files using a URL list

No AWS keys required, but all files being copied must have a status of released.

```
import encode_utils.transfer_to_gcp as gcp
import encode_utils.connection as euc
conn = euc.Connection("prod")
# Create URL list file
url_file = conn.gcp_transfer_urllist(
     file_ids=["ENCFF385UTX"],
     filename="files_to_transfer.txt")

# Upload files_to_transfer.txt to your GCS bucket, or some other public place␣
↪accessible via HTTP/S.
# Suggested to use a txt extension for your file rathar than tsv so that it can be␣
↪opened in the
# browser (i.e. in GCP to obtain the URL).

transfer = gcp.Transfer(gcp_project="sigma-night-206802")
transfer_job = transfer.from_urllist(
     urllist="https://files_to_transfer.txt",
     gcp_bucket="nathankw1",
     description="test")
```

1.2.4 Running the script

Requires that the user has AWS key permissions on the ENCODE buckets and file objects.

```
eu_s3_to_gcp.py --dcc-mode prod \
                --file-ids ENCFF270SAL ENCFF861EEE \
                --gcpbucket nathankw1 \
                --gcpproject sigma-night-206802 \
                --description test
```

CHAPTER 2

Scripts

2.1 eu_register.py

Given a tab-delimited or JSON input file containing one or more records belonging to one of the profiles listed on the ENCODE Portal (such as https://www.encodeproject.org/profiles/biosample.json), either POSTS or PATCHES the records. The default is to POST each record; to PATCH instead, see the `--patch` option.

When POSTING file records, the md5sum of each file will be calculated for you if you haven't already provided the *md5sum* property. Then, after the POST operation completes, the actual file will be uploaded to AWS S3. In order for this to work, you must set the *submitted_file_name* property to the full, local path to your file to upload. Alternatively, you can set *submitted_file_name* to and existing S3 object, i.e. s3://mybucket/reads.fastq.

Note that there is a special 'trick' defined in the `encode_utils.connection.Connection()` class that can be taken advantage of to simplify submission under certain profiles. It concerns the *attachment* property in any profile that employs it, such as the *document* profile. The trick works as follows: instead of constructing the *attachment* propery object value as defined in the schema, simply use a single-key object of the following format:

```
{"path": "/path/to/myfile"}
```

and the *attachment* object will be constructed for you.

```
usage: eu_register.py [-h] [-m DCC_MODE] [-d] [--no-aliases]
                      [--no-upload-file] -p PROFILE_ID -i INFILE [-w]
                      [-r REMOVE_PROPERTY] [--patch | --rm-patch]
```

2.1.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-d, --dry-run Set this option to enable the dry-run feature, such that no modifications are performed on the ENCODE Portal. This is useful if you'd like to inspect the logs or ensure the validity of your input file.

Default: False

--no-aliases Setting this option is NOT advised. Set this option for doing a POST when your input file doesn't contain an 'aliases' column, even though this property is supported in the corresponding ENCODE profile. When POSTING a record to a profile that includes the 'aliases' property, this package requires the 'aliases' property be used for traceability purposes and because without this property, it'll be very easy to create duplicate objects on the Portal. For example, you can easily create the same biosample as many times as you want on the Portal when not providing an alias.

Default: False

--no-upload-file Don't upload files when POSTing file objects

Default: False

-p, --profile_id The ID of the profile to submit to, i.e. use 'genetic_modification' for https://www.encodeproject.org/profiles/genetic_modification.json. The profile will be pulled down for type-checking in order to type-cast any values in the input file to the proper type (i.e. some values need to be submitted as integers, not strings).

-i, --infile The JSON input file or tab-delimited input file.

The tab-delimited file format: Must have a field-header line as the first line. Any lines after the header line that start with a '#' will be skipped, as well as any empty lines. The field names must be exactly equal to the corresponding property names in the corresponding profile. Non-scematic fields are allowed as long as they begin with a '#'; they will be skipped. If a property has an array data type (as indicated in the profile's documentation on the Portal), the array literals '[' and ']' are optional. Values within the array must be comma-delimited. For example, if a property takes an array of strings, then you can use either of these as the value:

1) str1,str2,str3
2) [str1,str2,str3]

On the other hand, if a property takes a JSON object as a value, then the value you enter must be valid JSON. This is true anytime you have to specify a JSON object. Thus, if you are submitting a genetic_modification and you have two 'introduced_tags' to provide, you can supply them in either of the following two ways:

1) {"name": "eGFP", "location": "C-terminal"},{"name": "FLAG","C-terminal"}
2) [{"name": "eGFP", "location": "C-terminal"},{"name": "FLAG","C-terminal"}]

The JSON input file Can be a single JSON object, or an array of JSON objects. Key names must match property names of an ENCODE record type (profile).

The following applies to either input file formats When patching objects, you must specify the 'record_id' field to indicate the identifier of the record. Note that this a special field that is not present in the ENCODE schema, and doesn't use the '#' prefix to mark it as non-schematic. Here you can specify any valid record identifier (i.e. UUID, accession, alias).

Some profiles (most) require specification of the 'award' and 'lab' attributes. These may be set as fields in the input file, or can be left out, in which case the default values for these attributes will be pulled from the environment variables DCC_AWARD and DCC_LAB, respectively.

-w, --overwrite-array-values Only has meaning in combination with the –patch option. When this is specified, it means that any keys with array values will be overwritten on the ENCODE Portal with the corresponding value to patch. The default action is to extend the array value with the patch value and then to remove any duplicates.

Default: False

-r, --remove-property Only has meaning in combination with the –rm-patch option. Properties specified in this argument will be popped from the record fetched from the ENCODE portal. Can specify as comma delimited string.

--patch Presence of this option indicates to PATCH an existing DCC record rather than register a new one.

Default: False

--rm-patch Presence of this option indicates to remove a property, as specified by the -r argument, from an existing DCC record, and then PATCH it with the payload specified in -i.

Default: False

2.2 eu_add_controlled_by

Sets the *File.controlled_by* property for all FASTQ File records on an Experiment.

An experiment can have multiple controls. This script will only work if:

1. The *Experiment.possible_controls* is already set, and
2. If more than one control, all controls have the same number of biologoical replicates, and
3. The number of biological replicates on the Experiment is the same as the number of biological replicates for any given control.

Having varying number of technical replicates is fine.

These requiements are set in order to have some meaningful, automated way of assigning control FASTQ files to experimental FASTQ files.

The algorithm works as follows:

1. The biological replicate numbers on the experiment are stored in a *list*, and ordered from least to greatest.
2. The biological replicate numbers on each control are stored in a *list*, and ordered from least to greatest.
3. Biological Replicate Assignment: For each control, its biological replicate number *list* created in step 2 is superimposed onto the *list* created in step 1. Assignment then takes place by positional matching. The matching is positional instead of simply by biological replicate number itself, since the numbering is known in some cases to not alwasy be sequential starting from 1.
4. Setting *possible_controls*: For each replicate in the Experiment, the forward reads FASTQ file linked to that replicate is assigned to the forward reads control FASTQ files. The control FASTQs are determined from the superimposition step above. For each control, all replicates that have the same *biologican_replicate_number* are included. If the sequencing is paired-end, then the same type of assignment happens for the reverse reads files.

```
usage: eu_add_controlled_by.py [-h] [-m DCC_MODE] -e EXP_ID
```

2.2.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-e, --exp-id An identifier of an Experiment record whose FASTQ File objects need to have their *controlled_by* property set.

2.3 eu_check_not_posted

Checks if the specified record identifiers are found on the Portal or not by doing a GET request on each one. If a 404 (not found) response is returned, then this identifier is written to the specified output file.

```
usage: eu_check_not_posted.py [-h] [-m DCC_MODE] -i INFILE -o OUTFILE
```

2.3.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-i, --infile Input file containing record identifiers, one per line. Any line starting with a '#' will be skipped.

-o, --outfile Output file containing a subset of the input file. Only record IDs that weren't found on the Portal will be written to this file, one per line.

2.4 eu_create_gcp_url_list

Creates a Google Storage Transfer Service URL list file, which can be used as input into the Google STS to transfer released ENCODE S3 files to your GCP buckets.

```
usage: eu_create_gcp_url_list.py [-h] [-m DCC_MODE]
                                 (-f FILE_IDS [FILE_IDS ...] | -i INFILE) -o
                                 OUTFILE
```

2.4.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-f, --file-ids An alternative to –infile, one or more ENCODE file identifiers. Don't mix ENCODE files from across buckets.

-i, --infile An alternative to –file-ids, the path to a file containing one or more file identifiers, one per line. Empty lines and lines starting with a '#' are skipped.

-o, --outfile The output URL list file name.

2.5 eu_generate_upload_creds.py

Generates upload credentials for one or more File records on the ENCODE Portal. The upload credentials store AWS authorization tokens to allow a user to upload a file to a presigned S3 URL.

The Exception *requests.exceptions.HTTPError* will be raised if the response from the server a not a successful status code for any File object when attempting to generate the upload credentials, and will utlimately halt the program with an error message.

```
usage: eu_generate_upload_creds.py [-h] [-m DCC_MODE]
                                   (-f FILE_IDS [FILE_IDS ...] | -i INFILE)
```

2.5.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-f, --file-ids One or more ENCODE File identifiers. They can be any valid ENCODE File object identifier, i.e. alias, uuid, accession, md5sum.

Default: []

-i, --infile **Input file containing File object identifiers, one** per line.

2.6 eu_get_aliases

Retrieves the aliases for the given set of DCC record identifiers.

```
usage: eu_get_aliases.py [-h] [-m DCC_MODE] -i INFILE -o OUTFILE
```

2.6.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-i, --infile Input file containing ENCODE object identifiers (one per line), i.e. UUID, accession, or @id. Empty lines and lines beginning with a '#' will be ignored.

-o, --outfile The output file, which is the same as the input file except for the addition of the tab-delimited columns - one for each alias.

2.7 eu_get_accessions

Use this script when there are records on the ENCODE Portal for which you know their aliases, but want to retreive their DCC accessions. This will only work if the record aliases you provided are registered with the records on the ENCODE Portal. Note that if the particular DCC profile at hand doesn't support the accession property, then the uuid will be returned.

```
usage: eu_get_accessions.py [-h] [-m DCC_MODE] -i INFILE -o OUTFILE
                            [-l SUBMITTER_LAB]
```

2.7.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-i, --infile Input file containing record aliases (one per line). Empty lines and lines beginning with a '#' will be ignored.

-o, --outfile The output file, which is in the same as the input file except for the addition of the tab-delimited columns - one for each alias.

-l, --submitter-lab The submitting lab alias prefix (i.e. michael-snyder) for these aliases. No need to set this option if your input file's aliases are already prefixed with the submitting lab. Furthermore, for any aliases lacking the prefix, the default will be taken from the DCC_LAB environment variable if not set here.

2.8 eu_get_replicate_fastq_encffs

Given an ENCODE experiment identifier, Prints the FASTQ ENCFF identifiers for the specified replicate and technical replicates, or all replicates. Also prints the replicate numbers. For each FASTQ identifer, the following is printed to stdout:

$BioNum_$TechNum_$ReadNumt$encff

where variables are defined as:

$BioNum - the biological repliate number $TechNum - the technial replicate number $ReadNum - '1' for a forwards reads FASTQ file, and '2' for a reverse reads FASTQ file.

```
usage: eu_get_replicate_fastq_encffs.py [-h] [-m DCC_MODE] -e EXP_ID
                                        [-b BIO_REP_NUM] [-t TECH_REP_NUM]
```

2.8.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-e, --exp-id The experiment to which the replicates belong. Must be set if –all-reps is absent.

-b, --bio-rep-num Print FASTQ ENCFFs for this specified biological replicate number.

-t, --tech-rep-num Print FASTQ ENCFFs for the specified technical replicate number of the specified biological replicate.

2.9 eu_get_replicate_numbers

Given an input file with replicate IDs, one per line, retrieves the attributes biological_replicate_number and technical_replicate_number. An output file is created in tab-delimited format with these two additional columns appended to the original lines.

```
usage: eu_get_replicate_numbers.py [-h] [-m DCC_MODE] -i INFILE -o OUTFILE
```

2.9.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-i, --infile Input file containing replicate identifiers (one per line), i.e. alias, UUID, or @id. Empty lines and lines starting with '#' will be skipped.

-o, --outfile The tab-delimited output file, which is the same as the input file except for the addition of the additional columns 'biological_replicate_number' and 'technical_replicate_number' (header-line not present).

2.10 eu_report_fastq_content_errors

Finds FASTQ files with a content error and indicates the error.

Given an input file containing experiment identifiers, one per row, looks up all FASTQ files on the experiment to check for content errors. If any FASTQ file object has a *status* property of "content_error", then the content error message will be extracted from the property *content_error_detail*. A new file will be output containing the error details.

```
usage: report_fastq_content_errors.py [-h] [-m DCC_MODE] -i INFILE -o OUTFILE
```

2.10.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-i, --infile Input file containing experiment identifiers, one per row. Any line starting with a '#' will be skipped.

-o, --outfile Output file containing three tab-delimited columns for each FASTQ file that is reported:

1. The accession of the Experiment record that the FASTQ file record belongs to.

2. The accession of the FASTQ file record.
3. The error message stored in the File objects *content_error_detail* property.

2.11 eu_s3_to_gcp.py

Copies one or more ENCODE files from AWS S3 storage to GCP storage by using the Google Storage Transfer Service. See *encode_utils.transfer_to_gcp.Transfer* for full documentation.

Note: Currently, only priviledged users with appropriate DCC API keys will be able to make use of this script because the Google STS requires that the source buckets be publicly discoverable. Sice the encode bucket policies deny the action s3:GetBucketLocation on the public principal. Non-priviledged users may find the alternative script *eu_create_gcp_url_list.py* to be a solution.

```
usage: eu_s3_to_gcp.py [-h] [-m DCC_MODE]
                       (-f FILE_IDS [FILE_IDS ...] | -i INFILE)
                       [-gb GCPBUCKET] -gp GCPPROJECT [-d DESCRIPTION]
                       [-c S3CREDS]
```

2.11.1 Named Arguments

-m, --dcc-mode The ENCODE Portal site ('prod' or 'dev', or an explicit host name, i.e. 'demo.encodedcc.org') to connect to.

-f, --file-ids An alternative to –infile, one or more ENCODE file identifiers. Don't mix ENCODE files from across buckets.

-i, --infile An alternative to –file-ids, the path to a file containing one or more file identifiers, one per line. Empty lines and lines starting with a '#' are skipped.

-gb, --gcpbucket The name of the GCP bucket.

-gp, --gcpproject The GCP project that is associated with gcp_bucket.

-d, --description The description to show when querying transfers via the Google Storage Transfer API, or via the GCP Console. May be left empty, in which case the default description will be the value of the first S3 file name to transfer.

-c, --s3creds AWS credentials. Provide them in the form *AWS_ACCESS_KEY_ID:AWS_SECRET_ACCESS_KEY*. Ideally, they'll be stored in the environment in variables by the same names. However, for additional flexability you can specify them here as well.

CHAPTER 3

Client API Modules

3.1 encode_utils.aws_storage

class encode_utils.aws_storage.**S3Upload**(*bucket_name*, *acl='public-read'*, *key_path=''*)
Bases: object

Simplifies the process of uploading files to a bucket in a specific location with the specified acl.

Parameters

- **bucket_name** – *str*. The name of the bucket to upload files to, i.e. pulsar-encode-assets.
- **acl** – *str*. See possible values at https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.put_object.
- **key_path** – *str*. The directory path in the specified bucket to upload all files to.

upload(*filename*)

Returns The key/path of the newly created object.

Return type *str*

class encode_utils.aws_storage.**S3Object**(*bucket_name=''*, *key=''*, *s3_uri=''*)
Bases: object

Represents an object in a S3 bucket. Internally used for calculating the md5sum and file size when submitting files to the ENCODE Portal.

You must set the appropriate AWS keys as documented in the wiki.

Parameters

- **bucket_name** – *str*. The name of the S3 bucket that contains your file of interest. For example, "mybucket".
- **key** – *str*. The object path in the bucket. For example, /path/to/reads.fastq.gz.

- **s3_uri** – Fully qualified path to the object in the bucket, i.e. s3://pulsar-lims-assets/path/to/reads.fastq.gz. If this is set, then the *buket_name* and *key* parameters are ignored since they will be set internally by parsing the value of *s3_uri*.

md5sum()
Retrieves the ETag as the md5sum. As explained here, an object's ETag may or may not be equal to its MD5 digest. In most cases, however, it will be. If ETags on your objects aren't equal, then this method shouldn't be used.

size()
Fetches the size of the S3 object by reading its content_length attribute.

3.2 encode_utils.connection

encode_utils.connection.**LOG_DIR = 'EU_Logs'**
The directory that contains the log files created by the *Connection* class.

class encode_utils.connection.**Connection**(*dcc_mode=None*, *dry_run=False*, *submission=False*, *no_log_file=False*)
Bases: object

Handles communication with the Portal regarding data submission and retrieval.

For data submission or modification, and working with non-released datasets, you must have the environment variables *DCC_API_KEY* and *DCC_SECRET_KEY* set. Check with your DCC data wrangler if you haven't been assigned these keys.

There are three log files opened in append mode in the directory specified by connection.LOG_DIR that are specific to whichever Portal you are connected to. When connected to Production, each log file name will include the token '_prod_'. For Development, the token will be '_dev_'. The three log files are named accordingly in reference to their purpose, and are classified as:

1. debug log file - All messages sent to STDOUT are also written to this log file. In addition, all messages written to the error log file described below are logged here.
2. error log file - Only terse error messages are sent to this log file for quick scanning of any potential issues. If you identify an error here that needs more explanation, then you should consult the debug log file.
3. posted log file - Tabulates what was successfully POSTED. There are three tab-delimited columnns ordered as submission timestamp, record alias, and record accession (or UUID if the *accession* property doesn't exist for the profile of the record at hand). Note that if a record has several aliases, then only the first one in the list for the *aliases* property is used.

ENCID_KEY = '_enc_id'
Identifies the name of the key in the payload that stores a valid ENCODE-assigned identifier for a record, such as alias, accession, uuid, md5sum, ... depending on the object being submitted. This is not a valid property of any ENCODE object schema, and is used in the patch() instance method to designate the record to update.

PROFILE_KEY = '_profile'
Identifies the name of the key in the payload that stores the ID of the profile to submit to. Like ENCID_KEY, this is a non-schematic key that is used only internally.

POST = 'post'
Constant

PATCH = 'patch'
Constant

debug_logger = None
A reference to the *debug* logging instance that was created earlier in `encode_utils.debug_logger`. This class adds a file handler, such that all messages sent to it are logged to this file in addition to STDOUT.

dcc_modes = None
An indication of which Portal instance to use. Set to 'prod' for the production Portal, and 'dev' for the development Portal. Alternatively, you can set an explicit host, such as demo.encodedcc.org. Leaving the default of None means to use the value of the *DCC_MODE* environment variable.

dry_run = None
Set to True to prevent any server-side changes on the ENCODE Portal, i.e. PUT, POST, PATCH, DELETE requests will not be sent to the Portal. After-POST and after-PATCH hooks (see the instance method *after_submit_hooks()*) will not be run either in this case. You can turn off this dry-run feature by calling the instance method *set_live_run()*.

error_logger = None
A `logging` instance with a file handler for logging terse error messages. The log file resides locally within the directory specified by the constant `connection.LOG_DIR`. Accepts messages >= `logging.ERROR`.

post_logger = None
A `logging` instance with a file handler for logging successful POST operations. The log file resides locally within the directory specified by the constant `connection.LOG_DIR`. Accepts messages >= `logging.INFO`.

auth
Sets the API and secret keys to use when authenticating with the DCC servers. These are determined from the values of the *DCC_API_KEY* and *DCC_SECRET_KEY* environment variables via the `_get_api_keys_from_env()` private instance method.

set_submission(*status*)
Sets the boolean value of the `self.submission` attribute.

Parameters **status** – *bool*.

check_dry_run()
Checks if the dry-run feature is enabled, and if so, logs the fact. This is mainly meant to be called by other methods that are designed to make modifications on the ENCODE Portal.

Returns The dry-run feature is enabled. *False*: The dry-run feature is turned off.

Return type *True*

set_dry_run()
Enables the dry-run feature and logs the fact.

set_live_run()
Disables the dry-run feature and logs the fact.

log_error(*msg*)
Sends 'msg' to both `self.error_logger` and `self.debug_logger`.

add_alias_prefix(*aliases*, *prefix=False*)
Given a list of aliases, adds the lab prefix to each one that doesn't yet have a prefix set. The lab prefix is taken as the passed-in *prefix*, otherwise, it defaults to the *DCC_LAB* environment variable, and it must be a value assigned by the DCC, i.e. "michael-snyder" for the Snyder Production Center. The DCC requires that aliases be prefixed in this manner.

Parameters

- **aliases** – *list* of aliases.

- **prefix** – *str*. The DCC assigned lab prefix to use. If not specified, then the default is the value of the DCC_LAB environment variable.

Returns *list*.

Raises *Exception* – A passed-in alias doesn't have a prefix set, and the default prefix could not be determined.

Examples:

```
add_alias_prefix(aliases=["my-alias"],prefix="michael-snyder")
# Returns ["michael-snyder:my-alias"]

add_alias_prefix(aliases=["michael-snyder:my-alias"],prefix="michael-snyder")
# Returns ["michael-snyder:my-alias"]

add_alias_prefix(aliases=["my_alias"], prefix="bad-value")
# Raises an Exception since this lab prefix isn't from a registered source␣
↪record on
# the Portal.
```

get_aliases(*dcc_id*, *strip_alias_prefix=False*)

Given an ENCODE identifier for an object, performs a GET request and extracts the aliases.

Parameters

- **dcc_id** – *str*. The ENCODE ID for a given object, i.e ENCSR999EHG.
- **strip_alias_prefix** – *bool*. *True* means to remove the alias prefix if all return aliases.

Returns The aliases.

Return type *list*

indexing()

Indicates whether the Portal is updating its schematic indicies.

Returns True if the Portal is indexing, False otherwise.

Return type *bool*

make_search_url(*search_args*)

Creates a URL encoded URL given the search arguments.

Parameters **search_args** – *list* of two-item tuples of the form `[(key, val), (key, val), ...]`.

Returns The URL containing the URL encoded query.

Return type *str*

search(*search_args=[]*, *url=None*, *limit=None*)

Searches the Portal using the provided query parameters, which will first be URL encoded. The user can pass in the query parameters and values via the *search_args* argument, or pass in a URL directly that contains a query string via the *url* argument, or provide values for both arguments in which case the query parameters specified in *search_args* will be added to the query parameters given in the URL.

If `self.submission == True`, then the query will be searced with "datastore=database", unless the 'database' query parameter is already set.

Parameters

- **search_args** – *list* of two-item tuples of the form `[(key, val), (key, val), ...]`. To support a != style query, append "!" to the key name.
- **url** – *str*. A URL used to search for records interactively in the ENCODE Portal. The query will be extracted from the URL.
- **limit** – *int*. The number of search results to send from the server. The default means to return all results.

Returns The search results.

Return type *list*

Raises *requests.exceptions.HTTPError* – The status code is not ok and != 404.

get_profile_from_payload(*payload*)

Useful to call when doing a POST (and `self.post()` does call this). Ensures that the profile key identified by `self.PROFILE_KEY` exists in the passed-in payload and that the value is a recognized ENCODE object profile (schema) identifier. Alternatively, the user can set the profile in the more convoluted *@id* property.

Parameters **payload** – *dict*. The intended object data to POST.

Returns The ID of the profile if all validations pass, otherwise.

Return type *str*

Raises

- `encode_utils.exceptions.ProfileNotSpecified` – Both keys `self.PROFILE_KEY` and *@id* are missing in the payload.
- *encode_utils.profiles.UnknownProfile* – The profile ID isn't recognized by the class *encode_utils.profiles.Profile*.

get_lookup_ids_from_payload(*payload*)

Given a payload to submit to the Portal, extracts the identifiers that can be used to lookup the record on the Portal, i.e. to see if the record already exists. Identifiers are extracted from the following fields:

1. `self.ENCID_KEY`,
2. aliases,
3. md5sum (in the case of a file object)

Parameters **payload** – *dict*. The data to submit.

Returns The possible lookup identifiers.

Return type *list*

get(*rec_ids*, *database=False*, *ignore404=True*, *frame=None*)

GET a record from the Portal.

Looks up a record in the Portal and performs a GET request, returning the JSON serialization of the object. You supply a list of identifiers for a specific record, and the Portal will be searched for each identifier in turn until one is either found or the list is exhausted.

Parameters

- **rec_ids** – *str* or *list*. Must be a *list* if you want to supply more than one identifier. For a few example identifiers, you can use a uuid, accession, . . . , or even the value of a record's *@id* property.

- **database** – *bool*. If True, then search the database directly instead of the Elasticsearch. indices. Always True when in submission mode (*self.submission* is True).
- **frame** – *str*. A value for the frame query parameter, i.e. 'object', 'edit'. See https://www.encodeproject.org/help/rest-api/ for details.
- **ignore404** – *bool*. Only matters when none of the passed in record IDs were found on the Portal. In this case, If set to *True*, then an empty *dict* will be returned. If set to *False*, then an Exception will be raised.

Returns The JSON response. Will be empty if no record was found AND `ignore404=True`.

Return type *dict*

Raises

- *Exception* – If the server responds with a FORBIDDEN status.
- *requests.exceptions.HTTPError* – The status code is not ok, and the cause isn't due to a 404 (not found) status code when `ignore404=True`.

set_attachment(*document*)

Sets the *attachment* property for any profile that supports it, such as *document* or *antibody_characterization*.

Checks if the provided file is an image in either of the JPEG or TIFF formats - if so, then checks the image orientation in the EXIF data and rotates if if necessary. The original image will not be modified.

Parameters **document** – *str*. A local file path.

Returns The 'attachment' propery value.

Return type *dict*

after_submit_file_cloud_upload(*rec_id*, *profile_id*)

An after-POST submit hook for uploading files to AWS.

Some objects, such as Files (*file.json* profile) need to have a corresponding file in the cloud. Where in the cloud the actual file should be uploaded to is indicated in File object's *file.upload_credentials.upload_url* property. Once the File object is posted, this hook is used to perform the actual cloud upload of the physical, local file represented by the File object.

Parameters

- **rec_id** – *str*. An identifier for the new File object on the Portal.
- **profile_id** – *str*. The ID of the profile that the record belongs to.

after_submit_hooks(*rec_id*, *profile_id*, *method=''*, *upload_file=True*)

Calls after-POST and after-PATCH hooks. This method is called from both the `post()` and `patch()` instance methods. Returns the None object immediately if the dry-run feature is enabled.

Some hooks only run if you are doing a PATCH, others if you are only doing a POST. Then there are some that run if you are doing either operation. Each hook that is called can potentially modify the payload.

Parameters

- **rec_id** – *str*. An identifier for the record on the Portal.
- **profile_id** – *str*. The profile identifier indicating the profile that the record belongs to.
- **method** – str. One of `self.POST` or `self.PATCH`, or the empty string to indicate which registered hooks to look through.
- **upload_file** – *bool*. If *False*, skip uploading files to the Portal. Defaults to *True*.

before_submit_alias(*payload*)

A pre-POST and pre-PATCH hook used to

1) Clean alias names by removing disallowed characters indicated by the DCC schema for the alias property.
2) Add the lab alias prefix to any aliases that are missing it. The *DCC_LAB* environment variable is consulted to fetch the lab name, and if not set then this will be a no-op.

Parameters **payload** – *dict*. The payload to submit to the Portal.

Returns The potentially modified payload.

Return type *dict*

before_submit_attachment(*payload*)

A pre-POST and pre-PATCH hook used to simplify the creation of an attachment in profiles that support it.

Checks the payload for the presence of the *attachment* property that is used by certain profiles, i.e. *document* and *antibody_characterization*, and then checks to see if a particular shortcut is being employed to indicate the attachment. That shortcut works as follows: if the dictionary value of the 'attachment' key has a key named 'path' in it (case-sensitive), then the value is taken to be the path to a local file. Then, the actual attachment object is constructed, as defined in the *document* profile, by calling `self.set_attachment()`. Note that this shortcut is particular to this `Connection` class, and when used the 'path' key should be the only key in the attachment dictionary as any others will be ignored.

Parameters **payload** – *dict*. The payload to submit to the Portal.

Returns The potentially modified payload.

Return type *dict*

before_post_file(*payload*)

A pre-POST hook that calculates and sets the *md5sum* property and *file_size* property for a file record. However, any of these two properties that is already set in the payload to a non-empty value will not be reset.

Parameters **payload** – *dict*. The payload to submit to the Portal.

Returns The potentially modified payload.

Return type *dict*

Raises `encode_utils.utils.MD5SumError` – Perculated through the function *encode_utils.utils.calculate_md5sum* when it can't calculate the md5sum.

before_post_fastq_file(*payload*)

A pre-POST hook for FASTQ file objects that checks whether certain rules are followed as defined in the file.json schema.

For example, if the FASTQ file is sequenced single-end, then the property `File.run_type` should be set to *single-ended* as expected, however, the property `File.paired_end` shouldn't be set in the payload, as the `File.run_type` property has the commment:

> Only paired-ended files should have paired_end values

before_submit_hooks(*payload*, *method=''*)

Calls pre-POST and pre-PATCH hooks. This method is called from both the `post()` and `patch()` instance methods.

Some hooks only run if you are doing a PATCH, others if you are only doing a POST. Then there are some that run if you are doing either operation. Each hook that is called can potentially modify the payload.

Parameters

- **payload** – *dict*. The payload to POST or PATCH.
- **method** – *str*. One of "post" or "patch", or the empty string to indicate which registered hooks to call. Some hooks are agnostic to the HTTP method, and these hooks are always called. Setting *method* to the empty string means to only call these agnostic hooks.

Returns The potentially modified payload that has been passed through all applicable pre-submit hooks.

Return type *dict*

post(*payload*, *require_aliases=True*, *upload_file=True*, *return_original_status_code=False*, *truncate_long_strings_in_payload_log=False*)

POST a record to the Portal.

Requires that you include in the payload the non-schematic key `self.PROFILE_KEY` to designate the name of the ENCODE object profile that you are submitting to, or the actual *@id* property itself.

If the *lab* property isn't present in the payload, then the default will be set to the value of the *DCC_LAB* environment variable. Similarly, if the *award* property isn't present, then the default will be set to the value of the *DCC_AWARD* environment variable.

Before the POST is attempted, any pre-POST hooks are fist called; see the method `self.before_submit_hooks`). After a successfuly POST, any after-POST submit hooks are also run; see the method `self.after_submit_hooks`.

Parameters

- **payload** – *dict*. The data to submit.
- **require_aliases** – *bool*. *True* means that the 'aliases' property is to be required in *payload*. This is the default and it is highly recommended not to change this because it'll be easy to create duplicates on the server if accidentally POSTING the same payload again. For example, you can easily create the same biosample as many times as you want on the Portal when not providing an alias. Furthermore, submitting labs should include at least one alias per record being submitted to the Portal for traceabilty purposes in the submitting lab.
- **upload_file** – *bool*. If *False*, when POSTing files the file data will not be uploaded to S3, defaults to *True*. This can be useful if you have custom upload logic. If the files to upload are already on disk, it is recommmended to leave this with the default, which will use *aws s3 cp* to upload them.
- **return_original_status_code** – *bool*. Defaults to *False*. If *True*, then will return the original *requests.Response.status_code* of the initial post, in addition to the usual *dict* response.
- **truncate_long_strings_in_payload_log** – *bool*. Defaults to *False*. If *True*, then long strings (> 1000 characters) present in the payload will be truncated before being logged.

Returns The JSON response from the POST operation, or the existing record if it already exists on the Portal (where a GET on any of it's aliases, when provided in the payload, finds the existing record). If *return_original_status_code=True*, then will return a *tuple* of the above *dict* and an *int* corresponding to the status code on POST of the initial payload.

Return type *dict*

Raises

- `encode_utils.exceptions.AwardPropertyMissing` – The *award* property isn't present in the payload and there isn't a defualt set by the environment variable *DCC_AWARD*.
- `encode_utils.exceptions.LabPropertyMissing` – The *lab* property isn't present in the payload and there isn't a default set by the environment variable *DCC_LAB*.
- `encode_utils.exceptions.MissingAlias` – The argument 'require_aliases' is set to True and the 'aliases' property is missing in the payload or is empty.
- `requests.exceptions.HTTPError` – The return status is not ok.

Side effects: self.PROFILE_KEY will be popped out of the payload if present, otherwise, the key "@id" will be popped out. Furthermore, self.ENCID_KEY will be popped out if present in the payload.

patch(*payload*, *raise_403=True*, *extend_array_values=True*)

PATCH a record on the Portal.

Before the PATCH is attempted, any pre-PATCH hooks are fist called (see the method `self.before_submit_hooks()`). If the PATCH fails due to the resource not being found (404), then that fact is logged to both the debug and error loggers.

Parameters

- **payload** – *dict*. containing the attribute key and value pairs to patch. Must contain the key `self.ENCID_KEY` in order to indicate which record to PATCH.
- **raise_403** – *bool*. *True* means to raise a `requests.exceptions.HTTPError` if a 403 status (forbidden) is returned. If set to *False* and there still is a 403 return status, then the object you were trying to PATCH will be fetched from the Portal in JSON format as this function's return value.
- **extend_array_values** – *bool*. Only affects keys with array values. *True* (default) means to extend the corresponding value on the Portal with what's specified in the payload. *False* means to replace the value on the Portal with what's in the payload.

Returns

The JSON response from the PATCH operation, or an empty dict if the record doesn't exist on the Portal. Will also be an empty dict if the dry-run feature is enabled.

Return type *dict*

Raises

- `KeyError` – The payload doesn't have the key `self.ENCID_KEY` set AND there aren't any aliases provided in the payload's 'aliases' key.
- `requests.exceptions.HTTPError` – if the return status is not ok (excluding a 403 status if 'raise_403' is False.

remove_props(*rec_id*, *props=[]*)

Runs a PUT request to remove properties of interest on the specified record.

Note that before-submit and after-submit hooks are not run here as they would be in *self.patch()* or *self.post()* (*before_submit_hooks()* and *after_submit_hooks()* are not called).

Parameters

- **rec_id** – *str*. An identifier for the record on the Portal.
- **props** – *list*. The properties to remove from the record.

Raises:

Returns *dict*. Contains the JSON returned from the PUT request.

remove_and_patch(*props*, *patch*, *raise_403=True*, *extend_array_values=True*)
Runs a PUT request to remove properties and patch a record in one request.

In general, this is a method combining `remove_props` and `patch`. This is useful because some schema dependencies requires property removal and property patch (including adding new properties) happening at the same time. Please note that after the record is retrieved from the portal, `props` will be removed before the `patch` is applied.

Parameters

- **props** – *list*. The properties to remove from the record.
- **patch** – *dict*. containing the attribute key and value pairs to patch. Must contain the key `self.ENCID_KEY` in order to indicate which record to PATCH.
- **raise_403** – *bool*. *True* means to raise a `requests.exceptions.HTTPError` if a 403 status (forbidden) is returned. If set to *False* and there still is a 403 return status, then the object you were trying to PATCH will be fetched from the Portal in JSON format as this function's return value.
- **extend_array_values** – *bool*. Only affects keys with array values. *True* (default) means to extend the corresponding value on the Portal with what's specified in the payload. *False* means to replace the value on the Portal with what's in the payload.

Returns

The JSON response from the PUT operation, or an empty dict if the record doesn't exist on the Portal. Will also be an empty dict if the dry-run feature is enabled.

Return type *dict*

Raises `requests.exceptions.HTTPError` – if the return status is not ok (excluding a 403 status if 'raise_403' is False).

send(*payload*, *error_if_not_found=False*, *extend_array_values=True*, *raise_403=True*)
Deprecated since version 1.1.1: Will be removed in the next major release.

A wrapper over `self.post()` and `self.patch()` that determines which to call based on whether the record exists on the Portal. Especially useful when submitting a high-level object, such as an experiment which contains many dependent objects, in which case you could have a mix where some need to be POST'd and some PATCH'd.

Parameters

- **payload** – *dict*. The data to submit.
- **error_if_not_found** – *bool*. If set to *True*, then a PATCH will be attempted and a `requests.exceptions.HTTPError` will be raised if the record doesn't exist on the Portal.
- **extend_array_values** – *bool*. Only matters when doing a PATCH, and Only affects keys with array values. *True* (default) means to extend the corresponding value on the Portal with what's specified in the payload. *False* means to replace the value on the Portal with what's in the payload.
- **raise_403** – *bool*. Only matters when doing a PATCH. *True* means to raise an requests.exceptions.HTTPError if a 403 status (forbidden) is returned. If set to *False* and there still is a 403 return status, then the object you were trying to PATCH will be fetched

from the Portal in JSON format as this function's return value (as handled by `self.patch()`).

Raises `requests.exceptions.HTTPError` – You want to do a PATCH (indicated by setting `error_if_not_found=True`) but the record isn't found.

get_fastqfiles_on_exp(*exp_id*)
Returns a list of all FASTQ file objects in the experiment.

Parameters **exp_id** – *str*. An Experiment identifier.

Returns Each element is the JSON form of a FASTQ file record.

Return type *list*

get_fastqfile_replicate_hash(*exp_id*)
Given a DCC experiment ID, gets its JSON representation from the Portal and looks in the *original* property to find FASTQ file objects and creates a *dict* organized by replicate numbers. Keying through the *dict* by replicate numbers, you can get to a particular file object's JSON serialization.

Parameters **exp_id** – *str*. An Experiment identifier.

Returns *dict* where each key is a biological_replicate_number. The value of each key is another *dict* where each key is a technical_replicate_number. The value of this is yet another *dict* with keys being file read numbers - 1 for forward reads, 2 for reverse reads. The value for a given key of this most inner dictionary is a list of JSON-serialized file objects.

Return type *dict*

extract_aws_upload_credentials(*creds*)
Sets values for the AWS CLI security credentials (for uploading a file to AWS S3) to the credentials found in a file record's *upload_credentials* property. The security credentials are stored in a *dict* where the keys are named after environment variables to be used by the AWS CLI.

Parameters **creds** – *dict*: The value of a File object's *upload_credentials* property.

Returns

dict containing keys named after AWS CLI environment variables being:

1. AWS_ACCESS_KEY_ID,
2. AWS_SECRET_ACCESS_KEY,
3. AWS_SECURITY_TOKEN,
4. UPLOAD_URL

Will be empty if the *upload_credentials* property isn't present in *file_json*.

Return type *dict*

get_upload_credentials(*file_id*)
Similar to `self.extract_aws_upload_credentials()`, but it goes a step further in that it is capable of regenerating the upload credentials if they aren't currently present in the file record.

Parameters **file_id** – *str*. A file object identifier (i.e. accession, uuid, alias, md5sum).

Returns

The value of the *upload_credentials* property if present, otherwise, the *dict* returned by `self.regenerate_aws_upload_creds`, which tries to generate the value for this property.

Return type *dict*

regenerate_aws_upload_creds (*file_id*)
Reissues AWS S3 upload credentials for the specified file record.

Parameters **file_id** – *str*. An identifier for a file record on the Portal.

Returns *dict* containing the value of the 'upload_credentials' key in the JSON serialization of the file record represented by *file_id*. Will be empty if new upload credentials could not be issued.

Return type *dict*

Raises *requests.exceptions.HTTPError* – The response from the server isn't a successful status code.

gcp_transfer_urllist (*file_ids*, *filename*)
Creates a "URL list" file to be used by the Google Storage Transfer Service (STS); see documentation at https://cloud.google.com/storage-transfer/docs/create-url-list. Once the URL list is created, you need to upload it somewhere that Google STS can reach it via HTTP or HTTPS. I recommend uploading the URL list to your GCS bucket. From there, you can get an HTTPS URL for it by clicking on your file name (while in the GCP Console) and then copying the URL shown in your Web browser, which can in turn be pasted directly in the Google STS.

Parameters

- **file_ids** – *list* of file identifiers. The corresponding S3 objects must have public read permission as required for the URL list.
- **filename** – *str*. The output filename in TSV format, which can be fed into the Google STS.

gcp_transfer_from_aws (*file_ids*, *gcp_bucket*, *gcp_project*, *description=''*, *aws_creds=()*)
Copies one or more ENCODE files from AWS S3 storage to GCP storage by using the Google STS. This is similar to the *gcp_transfer_urllist()* method - the difference is that S3 object paths are copied directly instead of using public HTTPS URIs, and AWS keys are required here.

See *encode_utils.transfer_to_gcp.Transfer()* for full documentation.

Parameters

- **file_ids** – *list*. One or more ENCODE files to transfer. They can be any valid ENCODE File object identifier. Don't mix ENCODE files from across buckets.
- **gcp_bucket** – *str*. The name of the GCP bucket.
- **gcp_project** – *str*. The GCP project that is associated with gcp_bucket. Can be given in either integer form or the user-friendly name form (i.e. sigma-night-206802)
- **description** – *str*. The description to show when querying transfers via the Google Storage Transfer API, or via the GCP Console. May be left empty, in which case the default description will be the value of the first S3 file name to transfer.
- **aws_creds** – *tuple*. Ideally, your AWS credentials will be stored in the environment. For additional flexability though, you can specify them here as well in the form `(AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY)`.

Returns The JSON response representing the newly created transferJob.

Return type *dict*

upload_file (*file_id*, *file_path=None*, *set_md5sum=False*)
Uploads a file to the Portal for the indicated file record. The file to upload can be specified by setting the *file_path* parameter, or by using the value of the ENCODE file profile's *submitted_file_name* property

of the given file object represented by the *file_id* parameter. The file to upload can be from any of the following sources:

1. Path to a local file,
2. S3 object, or
3. Google Storage object

For the AWS option above, the user must set the proper AWS keys, see the wiki documentation.

If the dry-run feature is enabled, then this method will return prior to launching the upload command.

Parameters

- **file_id** – *str*. An identifier of a *file* record on the ENCODE Portal.
- **file_path** – *str*. The local path to the file to upload, or an S3 object (i.e s3://mybucket/test.txt), or a Google Storage object (i.e. gs://mybucket/test.txt). If not set, defaults to *None* in which case the local file path will be extracted from the record's *submitted_file_name* property.
- **set_md5sum** – *bool*. True means to also calculate the md5sum and set the file record's *md5sum* property on the Portal (this currently is only implemented for local files and S3; not yet GCP). This will always take place whenever the property isn't yet set. Furthermore, setting to True will also cause the *file_size* property to be set. Normally these two properties would already be set as they are required in the *file* profile, however, if the wrong file was originally uploaded, then they must be reset when uploading a new file.

Raises `encode_utils.exceptions.FileUploadFailed` – The return code of the AWS upload command was non-zero.

get_platforms_on_experiment(*rec_id*)

Looks at all FASTQ files on the specified experiment, and tallies up the varying sequencing platforms that generated them. The platform of a given file record is indicated by the *platform* property. This is moreless used to verify that there aren't a mix of multiple different platforms present as normally all reads should come from the same platform.

Parameters **rec_id** – *str*. DCC identifier for an experiment.

Returns The de-duplicated list of platforms seen on the experiment's FASTQ files.

Return type *list*

post_document(*document*, *document_type*, *description*)

POSTS a document to the Portal.

The alias for the document will be the lab prefix plus the file name. The lab prefix is taken as the value of the *DCC_LAB* environment variable, i.e. 'michael-snyder'.

Parameters

- **document_type** – *str*. For possible values, see https://www.encodeproject.org/profiles/document.json. It appears that one should use "data QA" for analysis results documents.
- **description** – *str*. The description for the document.
- **document** – *str*. Local file path to the document to be submitted.

Returns The DCC UUID of the new document.

Return type *str*

download(*rec_id*, *get_stream=False*, *directory=None*)

Downloads the contents of the specified file or document object from the ENCODE Portal to either the

calling directory or the indicated download directory. The downloaded file will be named as it is on the Portal.

Alternativly, you can get a reference to the response object by setting the *get_stream* parameter to True. Useful if you want to inspect the response, i.e. see if there was a redirect and where to, or download the byte stream in a customized manner.

Parameters

- **rec_id** – *str*. A DCC identifier for a file or document record on the Portal.
- **directory** – *str*. The full path to the directory in which to download the file. If not specified, then the file will be downloaded in the calling directory.

Returns *str*. The full path to the downloaded file if the *get_stream* parameter is False. *requests.models.Response*: The *get_stream* parameter is True.

s3_object_path(*rec_id*, *url=False*)

Given an ENCODE File object's id (such as accession, uuid, alias), returns the full S3 object URI, or HTTP/HTTPS URI if url=True.

Parameters

- **rec_id** – *str*. A DCC object identifier of the record to link the document to.
- **url** – *bool*. True means to return the HTTP/HTTPS URI of the file rather than the S3 URI. Useful if this is a released file since you can download via the URL.

get_experiments_with_biosample(*rec_id*)

Returns all experiments that have a link to the given biosample record. Technically, there should only be at most one experiment linked to a given biosample, but it's possible that additional experiments can be, incorrectly, with audit flags going off.

Parameters **rec_id** – *str*. An identifier for a biosample record on the Portal.

Returns

***list* of dicts, where is dict is the JSON serialization of an experiment record that is** linked to the provided biosample record. If no experiments are linked, then this will be an empty list.

get_biosample_type(*classification*, *term_id=None*, *term_name=None*)

Searches the biosample_types for the given classification (i.e. tissue, cell line) and term_id or term_name. Both term_name and term_id need not be set - if both are than term_id will take precedence. The combination of classification and term_id/term_name uniquely identifies a biosample_type.

Parameters

- **classification** – *str*. A value for the 'classificaiton' property of the biosample_ontology profile.
- **term_id** – *str*. A value for the 'term_id' property of the biosample_ontology profile.
- **term_name** – *str*. A value for the 'term_name' property of the biosample_ontology profile.

Returns *dict*. Empty if not biosample_type found, otherwise the JSON representation of the record.

Raises

- *RecordNotFound* – No search results.
- *Exception* – More than one search result was returned. This should not happen and if it does then it's likely a bug on the server side.

3.3 encode_utils

An API and scripts for submitting datasetss to the ENCODE Portal.

encode_utils.**AWARD_PROP_NAME = 'award'**
The award property name that is common to all ENCODE Portal object profiles.

encode_utils.**ALIAS_PROP_NAME = 'aliases'**
The aliases property name that is common to almost all ENCODE Portal object profiles. Notably, the following profiles lack this property as of 2018-04-03: ['access_key_admin', 'publication', 'award', 'organism', 'page', 'image', 'user', 'lab']

encode_utils.**LAB_PROP_NAME = 'lab'**
The lab property name that is common to all ENCODE Portal object profiles.

encode_utils.**LAB = {}**
dict. Stores the lab property to the value of the environment variable *DCC_LAB* to serve as the default lab when submitting an object to the Portal. `encode_utils.connection.Connection.post()` will use this default if this property doesn't appear in the payload.

encode_utils.**LAB_PREFIX = ''**
str. Stores the prefix to add to each record alias when doing a POST operation. Most profiles have an 'alias' key, which stores a list of alias names that are useful to the lab. When POSTING objects to the Portal, these aliases must be prefixed with the lab name and end with a colon, and this configuration variable stores that prefix value.

encode_utils.**AWARD = {}**
dict. Stores the award property to the value of the environment variable *DCC_AWARD* to act as the default award when submiting an object to the Portal. `encode_utils.connection.Connection.post()` will use this default if this property doesn't appear in the payload, and the profile at hand isn't a member of the list `encode_utils.utils.Profile.AWARDLESS_PROFILES`.

encode_utils.**PROFILES_URL = 'profiles'**
The relative ENCODE Portal URL that points to all the profiles (schemas).

encode_utils.**DCC_MODES = {'dev': {'url': 'https://test.encodedcc.org'}, 'prod': {'url':**
A hash of known hosts one can connect to, where the key can be passed to the *dcc_mode* argument when instantiating the *connection.Connection* class.

encode_utils.**TIMEOUT = 60**
The timeout in seconds when making HTTP requests via the `requests` module.

encode_utils.**DEBUG_LOGGER_NAME = 'eu_debug'**
The name of the debug `logging` instance.

encode_utils.**ERROR_LOGGER_NAME = 'eu_error'**
The name of the error `logging` instance created in `encode_utils.connection.Connection()`, and referenced elsewhere.

encode_utils.**POST_LOGGER_NAME = 'eu_post'**
The name of the POST `logging` instance created in `encode_utils.connection.Connection()`, and referenced elsewhere.

encode_utils.**debug_logger = <Logger eu_debug (DEBUG)>**
A `logging` instance that logs all messages sent to it to STDOUT.

encode_utils.**error_logger = <Logger eu_error (ERROR)>**
A `logging` instance that accepts messages at the ERROR level.

3.4 encode_utils.profiles

Contains a `Profile` class for working with profiles on the ENCODE Portal. Note that the terms 'profile' and 'schema' are used interchangeably in this package.

encode_utils.profiles.**DEBUG_LOGGER = <Logger eu_debug.encode_utils.profiles (DEBUG)>**
A debug `logging` instance.

encode_utils.profiles.**ERROR_LOGGER = <Logger eu_error.encode_utils.profiles (ERROR)>**
An error `logging` instance.

exception encode_utils.profiles.**UnknownProfile**
Bases: `Exception`

Raised when the profile ID in question doesn't match any known profile ID.

class encode_utils.profiles.**Profiles**(*dcc_url*)
Bases: `object`

Encapsulates knowledge about the existing profiles on the Portal and contains useful methods for working with a given profile.

A defining purpose of this class is to validate the profile ID specified in a POST payload passed to `encode_utils.connection.Connection.post()`. This class is used to ensure that the profile specified there is a known profile on the Portal.

Parameters

- **dcc_url** – str. The portal URL being submitted to.
- **dcc_url** – *str*. The dcc_url as specified by Connection.dcc_mode.url.

FILE_PROFILE_ID = 'file'
Constant storing the *file.json* profile's ID. This is asserted for inclusion in `Profile.PROFILES`.

SUBMITTED_FILE_PROP_NAME = 'submitted_file_name'
Constant storing a property name of the *file.json* profile. The stored name is asserted for inclusion in the set of *File* properties.

MD5SUM_NAME_PROP_NAME = 'md5sum'
Constant storing a property name of the *file.json* profile. The stored name is asserted for inclusion in the set of *File* properties.

FILE_SIZE_PROP_NAME = 'file_size'
Constant sotring a property name of the *file.json* profile.

profiles
Constant (*dict*) set to the return value of the function `self.get_profiles()`. See documentation there for details.

profiles_with_property(*property_name*)
Returns a list of profile names that have a given property.

Parameters **property_name** – *str*. The name of the property.

Returns *list* of profile names.

get_profile_from_id(*at_id*)
Normalizes the *profile_id* so that it matches the format of the profile IDs stored in `self.profiles`, and ensures that the normalized profile ID is a member of this list.

Parameters **at_id** – *str*. An *@id* from the portal, e.g. */biosamples/ENCBS123ABC/*

Returns The normalized profile ID.

Return type *str*

Raises

- *UnknownProfile* – The normalized profile ID is not a member of the list
- *self.profiles*.

remove_duplicate_associations(*associations*)

Checks for duplicates in array properties containing string elements. Need to be careful as some cases can be tricky, i.e.

['/documents/id1', 'id1']

Such a duplicate should be identified and removed, leaving us with ["id1"].

Parameters **associations** – *list*.

Returns Deduplicated *list*.

3.5 encode_utils.transfer_to_gcp.py

Contains a Transfer class that encapsulates working with the Google Storage Transfer Service to transfer files from AWS S3 to GCP buckets. If you want to run this on a GCP VM, then in the command used to launch the VM you should specify an appropriate security account and the cloud-platform scope as the following example demonstrates:

```
gcloud compute instances create myinstance --service-account="test-819@sigma-night-
↪206802.iam.gserviceaccount.com" --scopes=cloud-platform --zone us-central1-a
```

Google implements OAuth 2 scopes for requesting accessing to specific Google APIs, and in our case it's the cloud-platform scope that we need, which is associated with the Storage Transfer API, amongst others. See the documentation in the *Transfer* class below for more details. Also, the Storage Transfer API documentation is available at https://developers.google.com/resources/api-libraries/documentation/storagetransfer/v1/python/latest/ https://cloud.google.com/docs/authentication/production#auth-cloud-explicit-python

If running this in Google Cloud Composer, you must use specific versions of several Google libraries so that the tasks running in the environment can properly use the credentials and scope that you delegated to the environment when creating it. This is a work-around, as the Composer environment is buggy at present in this regard. You'll need a requirements.txt file with the following (thanks to danxmoran for pointing out):

google-api-core==1.5.0 google-api-python-client==1.7.4 google-auth==1.5.1 google-auth-httplib2==0.0.3 google-cloud-core==0.28.1

Then use the following commands to create and set up your Cloud Composer environment:

‘‘‘ gcloud beta composer environments create test-cc-env3 –python-version=3 –location=us-central1 –zone=us-central1-a –disk-size=20GB –service-account=fasdf-29@sigma-night-206802.iam.gserviceaccount.com

gcloud composer environments update env3 –location us-central1 –update-pypi-packages-from-file requirements.txt

‘‘‘

exception encode_utils.transfer_to_gcp.**AwsCredentialsMissing**

Bases: Exception

Raised when a method needs AWS credentials but can't find them.

class encode_utils.transfer_to_gcp.**Transfer**(*gcp_project*, *aws_creds=()*)

Bases: object

See example at https://cloud.google.com/storage-transfer/docs/create-client and https://github.com/GoogleCloudPlatform/python-docs-samples/blob/master/storage/transfer_service/aws_request.py.

Encapsulates the transfer of files from AWS S3 storage to GCP storage by using the Google Storage Transfer Service (STS). The `create()` method is used to create a transfer job (termed *transferJob* in the STS API). A transferJob either runs once (a one-off job) or is scheduled to run repeatedly, depending on how the job schedule is specified.

Any transfer event of a trasferJob is termed as a transferOperation in the STS API. There are a few utility methods in this class that work with transferOperations.

You'll need to have a Google service account set up with at least the two roles below:

1) Project role with access level of Editor or greater.

2) Storage role with access level of Storage Object Creator or greater.

If running on a non-GCP VM, the service account credentials are fetched from the environment via the variable GOOGLE_APPLICATION_CREDENTIALS. This should be set to the JSON file provided to you by the GCP Console when you create a service account; see https://cloud.google.com/docs/authentication/getting-started for more details.

If instead you are running this on a GCP VM, then you should specify the service account and OAuth 2 scope when launching the VM as described at the beginning; there is no need use the service account file itself.

Note1: if this is the first time that you are using the Google STS on your GCP bucket, it won't work just yet as you'll get an error that reads:

> **Failed to obtain the location of the destination Google Cloud Storage (GCS) bucket due to insufficient permissions. Please verify that the necessary permissions have been granted. (Google::Apis::ClientError)**

To resolve this, I recommend that you go into the GCP Console and run a manual transfer there, as this adds the missing permission that you need. I personaly don't know how to add it otherwise, or even know what it is that's being added, but there you go!

Note2: If you try to transfer a file that is mistyped or doesn't exist in the source bucket, then this will not set a failed status on the transferJob. If you really need to know whether a file was tranferred in the API, you need to query the transferOperation; see the method *`get_transfers_from_job()`*.

Parameters

- **`gcp_project`** – *str*. The GCP project that contains your GCP bucket(s). Can be given in either integer form or the user-friendly name form (i.e. sigma-night-207122)
- **`aws_creds`** – *tuple*. Ideally, your AWS credentials will be stored in the environment. For additional flexability though, you can specify them here as well in the form `(AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY)`.

`from_s3`(*s3_bucket*, *s3_paths*, *gcp_bucket*, *overwrite_existing=False*, *description=''*)

Schedules an one-off transferJob that runs immediately to copy the specified file(s) from an s3_bucket to a gcp_bucket. AWS keys are required and must have the following permissions granted in source bucket policy:

1. s3:GetBucketLocation
2. s3:ListBucket
3. s3:GetObject

AWS Credentials are fetched from the environment via the variables *AWS_ACCESS_KEY_ID* and *AWS_SECRET_ACCESS_KEY*, unless passed explicitly to the aws_creds argument when instantiating the *Transfer* class.

Parameters

- **s3_bucket** – *str*. The name of the AWS S3 bucket.
- **s3_paths** – *list*. The paths to S3 objects in s3_bucket. Don't include leading '/' (it will be removed if seen at the beginning anyways). Up to 1000 files can be transferred in a given transfer job, per the Storage Transfer API transferJobs documentation. If you only need to transfer a single file, it may be given as a string.
- **gcp_bucket** – *str*. The name of the GCP bucket.
- **overwrite_existing** – *bool*. True means that files in GCP get overwritten by any files being transferred with the same name (key).
- **description** – *str*. The description to show when querying transfers via the Google Storage Transfer API, or via the GCP Console. May be left empty, in which case the default description will be the value of the first S3 file name to transfer.

Returns The JSON response representing the newly created transferJob.

Return type *dict*

from_urllist(*urllist*, *gcp_bucket*, *overwrite_existing=False*, *description=''*)

Schedules an one-off transferJob that runs immediately to copy the files specified in the URL list to GCS. AWS keys are not used, and all URIs must be publicliy assessible.

Parameters

- **gcp_bucket** – *str*. The name of the GCP bucket.
- **overwrite_existing** – *bool*. True means that files in GCP get overwritten by any files being transferred with the same name (key).
- **description** – *str*. The description to show when querying transfers via the Google Storage Transfer API, or via the GCP Console. May be left empty, in which case the default description will be the value of the first S3 file name to transfer.

Returns The JSON response representing the newly created transferJob.

Return type *dict*

get_transfers_from_job(*transferjob_name*)

Fetches descriptions in JSON format of any realized transfers under the specified transferJob. These are called transferOperations in the Google Storage Transfer API terminology.

See Google API example at https://cloud.google.com/storage-transfer/docs/create-manage-transfer-program?hl=ja in the section called "Check transfer operation status". See API details at https://cloud.google.com/storage-transfer/docs/reference/rest/v1/transferOperations.

Parameters **transferjob_name** – *str*. The value of the *name* key in the dictionary that is returned by self.from_s3 or self.from_urllist().

Returns

***list* of transferOperations belonging to the specified transferJob. This will be a list** of only a single element if the transferJob is a one-off transfer. But if this is a repeating transferJob, then there could be several transferOperations in the list.

get_transfer_status(*transferjob_name*)

Returns the transfer status of the first transferOperation that is returned for the given transferJob. Thus, this function really only makes sense for one-off transferJobs that don't repeat.

Note: if a transferJob attempts to transfer a non-existing file from the source bucket, this has no effect on the transferOperation status (it will not cause a FAILED status). Moreover, transferOperation status

doesn't look at what files were and were not transferred and is ony concerned with the execution status of the transferOperation job itself.

Parameters **transferjob_name** – *str*. The value of the *name* key in the dictionary that is returned by create().

3.6 encode_utils.utils

Contains utilities that don't require authorization on the DCC servers.

encode_utils.utils.**REQUEST_HEADERS_JSON = {'content-type': 'application/json'}**
Stores the HTTP headers to indicate JSON content in a request.

encode_utils.utils.**is_jpg_or_tiff**(*filename*)
Checks if the provided file is an image file that is formatted as either JPEG or TIFF.

Parameters **filename** – *str*. Local file.

Returns The provided file is not a JPEG or TIFF image. *str*: 'JPEG' if this is a JPEG image, or 'TIFF' if this is a TIFF image.

Return type *False*

Raises *OSError* – The provided file isn't a recognized image format.

encode_utils.utils.**orient_jpg**(*image*)
Given a JPG or TIFF, attempts to read the EXIF data to determine the orientation and then transform the image if needed. This function is called in *connection.Connection.set_attachment()*.

EXIF - exchangeable image file format - is only supported by JPG and TIFF formatted images. Such images aren't even required to set EXIF metadata. Imaging software sometimes sets EXIF to allow clients to read metadata such as what software took the picture, and what orientation it's in. This function is concerned with the oriention being in an upright position.

Note! Existing EXIF data will be lost for any transformed image. That's not a big issue for orientation, however, since software should consider the orientation to be 1 when EXIF isn't present anyways.

Parameters **image** – *str* or *bytes* instance. Use a string to supply the path to local JPG or TIFF file. Use a bytes object if you have the image data already in memory.

Returns

Dictionary with keys being:

1. from - int. The orientation that was read in, or 0 if unknown.
2. transformed - boolean. True if this function transformed the image, False otherwise. Note that False could either mean that the image didn't need any transformation or that the need for a transformation could not be determined based on EXIF metadata or lack thereof.
3. stream - A *bytes* instance.

Return type *dict*

Raises *InvalidExifOrientation* – The EXIF orientation data is present, but the orientation value isn't in the expected range of [1..8].

encode_utils.utils.**url_join**(*parts=[]*)
Useful for joining URL fragments to make a single cohesive URL, i.e. for searching. You can see several examples of its use in the *connection.Connection* class.

encode_utils.utils.**get_record_id**(*rec*)
Extracts the most suitable identifier from a JSON-serialized record on the ENCODE Portal. This is useful, for example, for other applications that need to store identifiers of specific records on the Portal. The identifier chosen is determined to be the 'accession' if that property is present, otherwise it's the first alias of the 'aliases' property is present, otherwise its the value of the 'uuid' property.

Parameters **rec** – *dict*. The JSON-serialization of a record on the ENCODE Portal.

Returns The extracted record identifier.

Return type *str*

Raises *Exception* – An identifier could not be extracted from the input record.

encode_utils.utils.**err_context**(*payload*, *schema*)
Validates the schema instance against the provided JSON schema.

Parameters

- **payload** – dict.
- **schema** – dict.

Returns *None* if there aren't any instance validation errors. Otherwise, a two-item tuple where the first item is the main error message; the second is a dictionary-based error hash that contains the contextual errors. This latter item may be empty.

encode_utils.utils.**calculate_md5sum**(*file_path*)
Calculates the md5sum for a local file or a S3 URI. If an S3 URI, the md5sum will be set as the objects ETag.

Parameters **file_path** – *str*. The path to a local file or an S3 URI, i.e. s3://bucket-name/key.

Returns The md5sum.

Return type *str*

Raises *FileNotFoundError* – The given file_path does not exist.

encode_utils.utils.**calculate_file_size**(*file_path*)
Calculates the file size in bytes for a local file or a S3 URI.

Parameters **file_path** – *str*. The path to a local file or an S3 URI, i.e. s3://bucket-name/key.

Returns *int*.

Raises *FileNotFoundError* – The given file_path does not exist.

encode_utils.utils.**print_format_dict**(*dico*, *indent=2*, *truncate_long_strings=False*)
Formats a dictionary for printing purposes to ease visual inspection. Wraps the `json.dumps()` function.

Parameters

- **indent** – *int*. The number of spaces to indent each level of nesting. Passed directly to the `json.dumps()` function.
- **truncate_long_strings** – *bool*. Defaults to *False*. If *True*, then long strings will be truncated before the object is serialized.

encode_utils.utils.**truncate_long_strings_in_objects**(*obj*, *max_num_chars=1000*)
Recursively truncates long strings in JSON objects, useful for reducing size of log messsages containing payloads with attachments using data URLs.

Parameters

- **obj** – Any type supported by *json.dump*, usually called with a *dict*.
- **max_num_chars** – *int*. The number of characters to truncate long strings to.

encode_utils.utils.**clean_aliases**(*aliases*)

Removes unwanted characters from the alias name. This function replaces:

-both '/' and '' with '_'. -# with "", as it is not allowed according to the schema.

Can be called prior to registering a new alias if you know it may contain such unwanted characters. You would then need to update your payload with the new alias to submit.

Parameters **aliases** – *list*. One or more record alias names to submit to the Portal.

Returns The cleaned alias.

Return type *str*

Example:: clean_alias_name("michael-snyder:a/troublesomelias") # Returns michael-snyder:a_troublesome_alias

encode_utils.utils.**create_subprocess**(*cmd*, *check_retcode=True*)

Runs a command in a subprocess and checks for any errors.

Creates a subprocess via a call to `subprocess.Popen` with the argument `shell=True`, and pipes stdout and stderr.

Parameters

- **cmd** – *str*. The command to execute.
- **check_retcode** – *bool*. When *True*, then a `subprocess.SubprocessError` is raised when the subprocess returns a non-zero return code. The error message will display the command that was executed along with its actual return code, as well as any messages that the subprocess sent to STDOUT and STDERR. When *False*, the `subprocess.Popen` instance will be returned instead and it is expected that the caller will call its `communicate` method.

Returns Two-item tuple containing the subprocess's STDOUT and STDERR streams' content if `check_retcode=True`, otherwise a `subprocess.Popen` instance.

Raises `subprocess.SubprocessError` – There is a non-zero return code and `check_retcode=True`.

encode_utils.utils.**strip_alias_prefix**(*alias*)

Splits *alias* on ':' to strip off any alias prefix. Aliases have a lab-specific prefix with ':' delimiting the lab name and the rest of the alias; this delimiter shouldn't appear elsewhere in the alias.

Parameters **alias** – *str*. The alias.

Returns The alias without the lab prefix.

Return type *str*

Example:

```
strip_alias_prefix("michael-snyder:B-167")
# Returns "B-167"
```

encode_utils.utils.**add_to_set**(*entries*, *new*)

Adds an entry to a list and makes a set for uniqueness before returning the list.

Parameters

- **entries** – *list*.
- **new** – (any datatype) The new member to add to the list.

Returns A deduplicated list.

Return type *list*

encode_utils.utils.**does_lib_replicate_exist**(*replicates_json*, *lib_accession*, *biological_replicate_number=False*, *technical_replicate_number=False*)

Regarding the replicates on the specified experiment, determines whether any of them belong to the specified library. Optional constraints are the 'biological_replicate_number' and the 'technical_replicate_number' props of the replicates.

Parameters

- **replicates_json** – *list*. The value of the *replicates* property of an Experiment record.
- **lib_accession** – *str*. The value of a library object's *accession* property.
- **biological_replicate_number** – int. The biological replicate number.
- **technical_replicate_number** – int. The technical replicate number.

Returns The replicate UUIDs that pass the search constraints.

Return type *list*

encode_utils.utils.**remove_duplicate_objects**(*objects*)

Checks for duplicates in array properties containing dictionary elements.

Parameters **objects** – *list*.

Returns Deduplicated *list*.

CHAPTER 4

Unit Tests

4.1 encode_utils.tests.test_utils

4.2 encode_utils.tests.test_connection

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

Index

R

S

T

U