
pyPreservica

James Carr

May 01, 2024

TABLE OF CONTENTS:

1	Why Should I Use This?	3
2	SDK Features	5
2.1	Entity API Features	5
2.2	Content API Features	5
2.3	Upload API Features	6
2.4	Admin API Features	6
2.5	Retention Management API Features	6
2.6	Workflow API Features	6
2.7	Webhook API Features	6
2.8	Authority Records API Features	7
3	Background	9
4	PIP Installation	13
5	Get the Source Code	15
6	Contributing	17
7	Support	19
8	Examples	21
9	Authentication	23
10	2 Factor Authentication	25
11	SSL Certificates	27
12	Application Logging	29
13	Entity API	31
13.1	Fetching Entities (Assets, Folders & Content Objects)	31
13.2	Fetching Children of Entities	32
13.3	Creating new Folders	34
13.4	Adding Physical Assets	34
13.5	Updating Entities	34
13.6	Security Tags	35
13.7	3rd Party External Identifiers	35
13.8	Descriptive Metadata	37
13.9	Relationships Between Entities	38

13.10	Representations, Content Objects & Generations	39
13.11	Integrity Check History	42
13.12	Moving Entities	42
13.13	Deleting Entities	42
13.14	Finding Updated Entities	43
13.15	Downloading Files	43
13.16	Events on Specific Entities	44
13.17	Events Across Entities	44
13.18	Ingest Events	44
13.19	Asset and Folder Thumbnail Images	44
13.20	Replacing Content Objects	45
13.21	Export OPEX Package	45
14	Content API	47
14.1	object-details	47
14.2	indexed-fields	48
14.3	Search	48
14.4	Search Progress	49
14.5	Reporting Examples	50
14.5.1	Create a spreadsheet containing all Assets within the repository	50
14.5.2	Create a spreadsheet containing all Assets and Folders within the repository	51
14.5.3	Create a spreadsheet containing all Assets and Folders underneath a specific folder	51
14.6	User Security Tags	52
15	Upload API	53
15.1	Uploading Packages	53
15.2	Monitoring Upload Progress	54
15.3	Creating Packages	55
15.4	Creating Packages with Multiple Representations	58
15.5	Custom Fixity Generation	58
15.6	Bulk Package Creation	59
15.7	Package Examples	60
15.7.1	Ingest a single digital file as an asset, with a progress bar during upload, delete the package after upload has completed.	60
15.7.2	Ingest a single digital file as an asset, with a custom asset Title and Description	60
15.7.3	Ingest each jpeg file in a directory as an individual asset	61
15.7.4	Ingest a single digital file as an asset with a 3rd party identifier and custom metadata	61
15.7.5	Create a single Asset with 2 Representations (Preservation and Access) each Representation has 1 Content Object	62
15.7.6	Create a package with 1 Asset 2 Representations (Preservation and Access) and multiple Content Objects (one for every image)	62
15.8	Spreadsheet Metadata	63
15.9	Ingest Web Video	64
15.10	Ingest Twitter Feeds	66
15.11	Crawl and ingest from a filesystem	66
16	Workflow API	67
16.1	Fetching Workflow Contexts	67
16.2	Fetching Workflow Instances	68
16.3	Starting Workflows	68
17	Admin API	71
17.1	Metadata Management (XSD Schema's, XML Documents & XSLT Transforms)	71
17.2	User Management	74
17.3	Security Tags	75

18 Retention API	77
18.1 Retention Policies	77
18.2 Retention Assignments	78
19 Registry API	79
19.1 Non-Authenticated Read Access	79
20 Monitor API	83
20.1 Monitors	83
20.2 Messages	83
20.3 Monitor Timeseries	84
21 WebHook API	85
21.1 Subscribing	85
21.2 Listing Subscriptions	86
21.3 Unsubscribe	86
21.4 Reference Web Server	86
22 Authority Records API	89
22.1 Authority Tables	89
22.2 Authority Records	90
23 Example Applications	93
24 Developer Interface	97
24.1 Entity API	97
24.2 Content API	110
24.3 Upload API	111
24.4 Retention Management API	115
24.5 Workflow API	117
24.6 Administration and Management API	119
24.7 Process Monitor API	124
24.8 WebHook API	125
24.9 Authority Records API	126
25 Index	129
Python Module Index	131
Index	133

Release v2.6.7.

pyPreservica is an open source, python client for the Preservica APIs

pyPreservica is a 3rd party Python Software Development Kit (SDK) for the Preservica API, which allows Preservica users to write software that makes use of the Preservica repository services. This library provides classes for working with a range of the Preservica APIs.

<https://developers.preservica.com/api-reference>

This version of the documentation is for use against a Preservica 7.0-6.2 systems For Preservica 6.0 and 6.1 see [the previous version](#)

pyPreservica is an open source 3rd party library and is not affiliated with [Preservica Ltd](#) There is no support for use of the library by Preservica Ltd. For support see [Support](#)

WHY SHOULD I USE THIS?

The goal of pyPreservica is to allow you to make use of the Preservica Entity API for reading and writing objects within a Preservica repository without having to manage the underlying REST HTTPS requests and XML parsing. The library provides a level of abstraction which reflects the underlying data model, such as structural and information objects.

The pyPreservica library allows Preservica users to build applications which interact with the repository such as meta-data synchronisation with 3rd party systems etc.

Hint: Access to the Preservica API's for the cloud hosted system does depend on which Preservica Edition has been licensed. See <https://preservica.com/digital-archive-software/products-editions> for details.

SDK FEATURES

2.1 Entity API Features

- Fetch and Update Entity Objects (Folders, Assets, Content Objects)
- Add, Delete and Update External Identifiers
- Add, Delete and Update Descriptive Metadata Fragments
- Change Security tags on Folders and Assets
- Create new Folder Entities
- Move Assets and Folders within the repository
- Deleting Assets and Folders
- Fetch Folders and Assets belonging to parent Folders
- Retrieve Representations, Generations & Bitstreams from Assets
- Download digital files and thumbnails
- Fetch lists of changed entities over the last n days
- Request information on completed integrity checks
- Add or remove asset and folder icons
- Replace existing content objects within an Asset
- Export OPEX Package
- Fetch audit trail events on Entities and across the repository
- Create Relationships between Assets

2.2 Content API Features

- Fetch a list of indexed Solr Fields
- Search based on a single query term
- Filtered searches on indexed fields

2.3 Upload API Features

- Create single Content Object Packages with multiple Representations
- Create multiple Content Object Packages with multiple Representations
- Upload packages to Preservica
- Spreadsheet Metadata
- Ingest Web Video
- Ingest Twitter Feeds

2.4 Admin API Features

- Schema Management (XML Templates, XSD Schema's & XSLT Transforms)
- User Management (create and remove user accounts)
- Security Tags (add and remove security tags)

2.5 Retention Management API Features

- Create new retention policies
- Delete retention policies
- Update retention policies
- Assign retention policies to entities

2.6 Workflow API Features

- Get Workflow Contexts
- Get Workflow Instance
- Start Workflow Instances

2.7 Webhook API Features

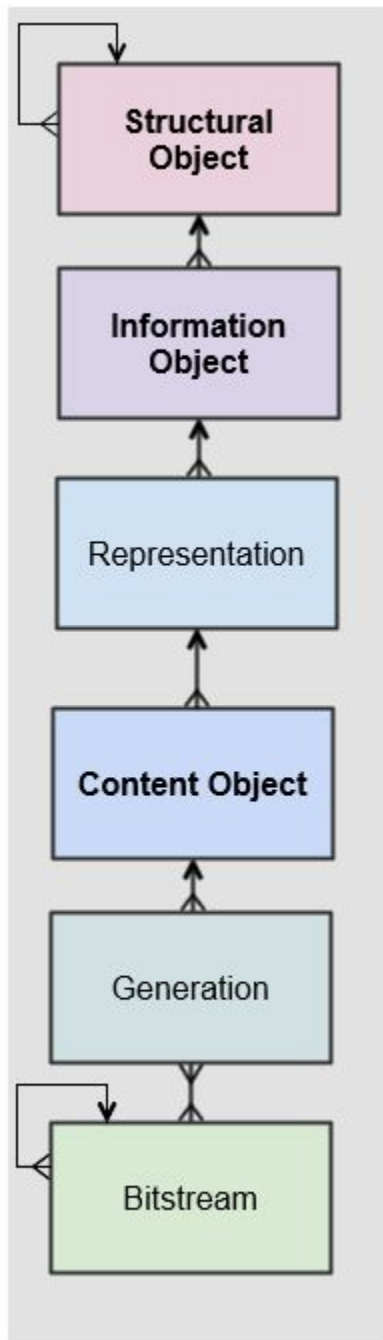
- Subscribe to Webhook endpoints
- Unsubscribe
- List Subscriptions

2.8 Authority Records API Features

- Get an Authority table by its reference
- List all Authority tables
- Return all records from a Authority table
- Add records to an Authority table
- Delete records from an Authority table

BACKGROUND

The key to working with the pyPreservica library is that the services follow the Preservica core data model closely.



The Preservica data model represents a hierarchy of entities, starting with the **structural objects** which are used to represent aggregations of digital assets. Structural objects define the organisation of the data. In a library context they may be referred to as collections, in an archival context they may be Fonds, Sub-Fonds, Series etc and in a records management context they could be simply a hierarchy of folders or directories.

These structural objects may contain other structural objects in the same way as a computer filesystem may contain folders within folders.

Within the structural objects comes the **information objects**. These objects which are sometimes referred to as the digital assets are what PREMIS defines as an Intellectual Entity. Information objects are considered a single intellectual

unit for purposes of management and description: for example, a book, document, map, photograph or database etc.

Representations are used to define how the information object are composed in terms of technology and structure. For example, a book may be represented as a single multiple page PDF, a single eBook file or a set of single page image files.

Representations are usually associated with a use case such as access or long-term preservation. All Information objects have a least one representation defined by default. Multiple representations can be either created outside of Preservica through a process such as digitisation or within Preservica through preservation processes such a normalisation.

Content Objects represent the components of the asset. Simple assets such as digital images may only contain a single content object whereas more complex assets such as books or 3d models may contain multiple content objects. In most cases content objects will map directly to digital files or bitstreams.

Generations represent changes to content objects over time, as formats become obsolete new generations may need to be created to make the information accessible.

Bitstreams represent the actual computer files as ingested into Preservica, i.e. the TIFF photograph or the PDF document.

PIP INSTALLATION

pyPreservica is available from the Python Package Index (PyPI)

<https://pypi.org/project/pyPreservica/>

pyPreservica is built and tested against Python 3.8. Older versions of Python may not work.

To install pyPreservica, simply run this simple command in your terminal of choice:

```
$ pip install pyPreservica
```

or you can install in a virtual python environment using:

```
$ pipenv install pyPreservica
```

pyPreservica is under active development and the latest version is installed using

```
$ pip install --upgrade pyPreservica
```


GET THE SOURCE CODE

pyPreservica is developed on GitHub, where the code is [always available](#).

You can clone the public repository

```
$ git clone git://github.com/carj/pyPreservica.git
```


CONTRIBUTING

Bug reports and pull requests are welcome on GitHub at <https://github.com/carj/pyPreservica>

SUPPORT

pyPreservica is 3rd party open source client and is not affiliated or supported by [Preservica Ltd](#)

For announcements about new versions and discussion of pyPreservica please subscribe to the google groups forum <https://groups.google.com/g/pypreservica>

Bug reports can be raised directly on either [GitHub](#) or on the google group forum

General questions and queries about using pyPreservica posted on the google group forum above.

EXAMPLES

Using the python console, create the entity API client object and request an Asset (Information Object) by its unique reference and display some of its attributes.

All entities within the Preservica system have one unique reference which can be used to retrieve them.

The reference used to fetch entities (Assets, Folders) is the Preservica internal unique identifier. This is a universally unique identifier (UUID)

You can find the reference when viewing the object metadata within Explorer. Later on we will look at how we can fetch entities using other 3rd party external identifiers which may be more meaningful such as ISBNs DOIs etc.

To create the client object you will need valid credentials to connect to the Preservica server. See the following section on available authentication options.

```
>>> from pyPreservica import *
>>> client = EntityAPI()
>>> client
pyPreservica version: 0.8.5 (Preservica 6.2 Compatible)
Connected to: us.preservica.com Version: 6.2.0 as test@test.com
>>> asset = client.asset("dc949259-2c1d-4658-8eee-c17b27a8823d")
>>> asset.reference
'dc949259-2c1d-4658-8eee-c17b27a8823d'
>>> asset.title
'LC-USZ62-20901'
>>> asset.parent
'ae108c8f-b058-4228-b099-6049175d2f0c'
>>> asset.security_tag
'open'
>>> asset.entity_type
<EntityType.ASSET: 'IO'>
```

If your credentials are valid, pyPreservica returns a client object which is the connection to the server. Printing the client returns information about the connection such as the server and the user name etc. This can be useful to check that you are connected to the correct system.

All entities have a parent reference attribute, for Assets this always points to the parent Folder. For Content Objects the parent points to the Asset and for Folders it points to the parent Folder if it exists. Folders at the root level of the repository do not have a parent and the attribute returns the special Python value of None

This example shows how pyPreservica can be used to upload and ingest a local file, picture.tiff into Preservica using the UploadAPI class. The tiff file will be ingested as a new Asset object inside the existing Preservica folder given by the folder UUID. The `simple_asset_package` function creates the package, in this case an XIPv6 formatted package and the `upload_zip_package` method uploads it directly to the Preservica server using the S3 protocol.

```
>>> from pyPreservica import *  
  
>>> client = UploadAPI()  
>>> folder = "dc949259-2c1d-4658-8eee-c17b27a8823d"  
>>> zip_p = simple_asset_package(preservation_file="picture.tiff", parent_folder=folder)  
>>> client.upload_zip_package(zip_p)
```

AUTHENTICATION

pyPreservica provides 4 different methods for authentication. The library requires the username and password of a Preservica user and an optional Tenant identifier along with the server hostname.

Tip: The Tenant parameter is now optional when connecting to a Preservica 6.3 system.

1 Method Arguments

Include the user credentials as arguments to the EntityAPI Class

```
from pyPreservica import *  
  
client = EntityAPI(username="test@test.com", password="123444",  
                  tenant="PREVIEW", server="preview.preservica.com")
```

If you don't want to include your Preservica credentials within your python script because you are sharing scripts or using a version control system then one of the following two methods should be used.

2 Environment Variable

Export the credentials as environment variables as part of the session

```
$ export PRESERVICA_USERNAME="test@test.com"  
$ export PRESERVICA_PASSWORD="123444"  
$ export PRESERVICA_TENANT="PREVIEW"  
$ export PRESERVICA_SERVER="preview.preservica.com"  
  
$ python3
```

```
from pyPreservica import *  
  
client = EntityAPI()
```

3 Properties File

Create a properties file called "credentials.properties" with the following property names and save to the working directory

```
[credentials]  
username=test@test.com  
password=123444  
tenant=PREVIEW  
server=preview.preservica.com
```

```
from pyPreservica import *  
  
client = EntityAPI()
```

You can create a new credentials.properties file automatically using the `save_config()` method

```
from pyPreservica import *  
  
client = EntityAPI(username="test@test.com", password="123444",  
                  tenant="PREVIEW", server="preview.preservica.com")  
client.save_config()
```

4 Shared Secrets

pyPreservica now supports authentication using shared secrets rather than a login account username and password. This allows a trusted external applications such as pyPreservica to acquire a Preservica API authentication token without having to use a set of login credentials.

This option is useful if you want to provide limited API access to a 3rd party without providing login access to Preservica.

To use the shared secret authentication you need to add a secure secret key to your Preservica system.

The username, password, tenant and server attributes are used as normal, the password field now holds the shared secret and not the users password.

```
from pyPreservica import *  
  
client = EntityAPI(username="test@test.com", password="shared-secret", tenant="PREVIEW",  
                  server="preview.preservica.com", use_shared_secret=True)
```

If you are using a credentials.properties file then

```
from pyPreservica import *  
  
client = EntityAPI(use_shared_secret=True)
```

2 FACTOR AUTHENTICATION

pyPreservica now supports the new 2-Factor authentication for APIs introduced with Preservica 6.8

The Preservica system should be first setup for 2-Factor authentication and the one time password key used to seed the 2FA (HMAC-Based One-Time Password Algorithm) should be retained and used with the API.

The one time password or seed key is available to view and should be saved when setting up the 2FA for a user. You can find the two factor seed key from the user 2FA setup page under the “Reveal Key” button at the bottom of the page.

Keep this key secret along with your account password as it will be required when authenticating the API calls.

Your administrator has enabled two factor authentication. You need to enter a token to continue.

000000

Enter Token

Before continuing, you will need to set up two factor authentication. To do this:

1. Install Google Authenticator on your phone (or an equivalent app)
2. Open the Google Authenticator app
3. Tap menu, then tap "Set up account" and "Scan a barcode"
4. Your phone will now be in a scanning mode. When you are in this mode, scan the QR code below.

If you are not able to scan a QR code, you can enter the key manually.

Reveal key

To call pyPreservica once 2-Factor authentication process has been setup, you need the username and password as normal along with the additional two factor key.

You can pass the additional two factor key as an argument to the constructor for the API classes or use environment variables or the credentials file.

```
from pyPreservica import *

client = EntityAPI(username="test@test.com", password="my-login-password", tenant=
↳ "PREVIEW",
                    server="preview.preservica.com", two_fa_secret_key=
↳ "AJC5DEGUV6UQ1TT")
```

The environment variable for holding the 2 factor seed key is called *PRESERVICA_2FA_TOKEN* and the credential file property name is *twoFactorToken*.

```
$ export PRESERVICA_2FA_TOKEN=AJC5DEGUV6UQ1TT
```

i.e

```
[credentials]
username=test@test.com
password=123444
tenant=PREVIEW
server=preview.preservica.com
twoFactorToken=AJC5DEGUV6UQ1TT
```

Tip: Preservica uses time based One Time Passwords (OTP), this means the time on your local machine must match time on the server.

SSL CERTIFICATES

pyPreservica will by default connect to servers which use the <https://> protocol and will always validate certificates when connected via https.

For Enterprise on Premise customers on secure networks, you can change the default protocol to use <http://> via the constructor.

```
client = EntityAPI(protocol="http")
```

pyPreservica uses the [Certifi](#) project to provide SSL certificate validation.

Self-signed certificates used by on-premise deployments are not part of the Certifi certification authority (CA) bundle and therefore need to be set explicitly.

The CA bundle is a file that contains root and intermediate certificates. The end-entity certificate along with a CA bundle constitutes the certificate chain.

For on-premise deployments the trusted CAs can be specified through the REQUESTS_CA_BUNDLE environment variable. e.g.

```
$ export REQUESTS_CA_BUNDLE=/usr/local/share/ca-certificates/my-server.cert
```


APPLICATION LOGGING

You can add logging to your pyPreservica scripts by simply including the following

```
import logging
from pyPreservica import *

logging.basicConfig(level=logging.DEBUG)

client = EntityAPI()
```

This will log all messages from level DEBUG or higher to standard output, i.e the console.

When logging to files, the main thing to be wary of is that log files need to be rotated regularly. The application needs to detect the log file being renamed and handle that situation. While Python provides its own file rotation handler, it is best to leave log rotation to dedicated tools such as logrotate. The WatchedFileHandler will keep track of the log file and reopen it if it is rotated, making it work well with logrotate without requiring any specific signals.

Here's a sample implementation.

```
import logging
import logging.handlers
import os

from pyPreservica import *

handler = logging.handlers.WatchedFileHandler("pyPreservica.log")
formatter = logging.Formatter(logging.BASIC_FORMAT)
handler.setFormatter(formatter)
root = logging.getLogger()
root.setLevel(logging.DEBUG)
root.addHandler(handler)

client = EntityAPI()
```


ENTITY API

Making a call to the Preservica repository is very simple.

Begin by importing the pyPreservica module at the start of the Python script. You can import only the API you need or the whole library.

To import all the pyPreservica functionality use:

```
from pyPreservica import *
```

Now, let's create the EntityAPI client object, this can have any name, but let's call it `client` to keep things simple.

```
client = EntityAPI()
```

The `client` object will manage the connection to the server and will be responsible for creating the API authentication tokens as needed.

13.1 Fetching Entities (Assets, Folders & Content Objects)

The following Python code examples show how data model entities, (Assets, Folders & Content Objects) can be returned from Preservica using their internal Preservica identifiers.

The following shows how you can fetch an Asset by its reference and then print its attributes to the screen.

```
from pyPreservica import *

asset = client.asset("9bad5acf-e7a1-458a-927d-2d1e7f15974d")
print(asset.reference)
print(asset.title)
print(asset.description)
print(asset.security_tag)
print(asset.parent)
print(asset.entity_type)
```

We can also fetch the same attributes for both Folders

```
folder = client.folder("0b0f0303-6053-4d4e-a638-4f6b81768264")
print(folder.reference)
print(folder.title)
print(folder.description)
print(folder.security_tag)
```

(continues on next page)

(continued from previous page)

```
print(folder.parent)
print(folder.entity_type)
```

and Content Objects

```
content_object = client.content_object("1a2a2101-6053-4d4e-a638-4f6b81768264")
print(content_object.reference)
print(content_object.title)
print(content_object.description)
print(content_object.security_tag)
print(content_object.parent)
print(content_object.entity_type)
```

Assets, Folders & Content Objects actually have a number of attributes in common, such as `title`, `description` etc. Technically they are all objects of type `Entity`.

We can fetch any of Assets, Folders and Content Objects using the entity type and the unique reference

```
asset = client.entity(EntityType.ASSET, "9bad5acf-e7a1-458a-927d-2d1e7f15974d")

folder = client.entity(EntityType.FOLDER, asset.parent)
```

To get a list of parent Folders of an Asset all the way to the root of the repository

```
asset = client.asset("9bad5acf-e7a1-458a-927d-2d1e7f15974d")

folder = client.folder(asset.parent)
print(folder.title)
while folder.parent is not None:
    folder = client.folder(folder.parent)
    print(folder.title)
```

13.2 Fetching Children of Entities

The immediate children of a Folder can also be retrieved using the library.

To get all the top level or root Folders use

```
for root_folder in client.descendants(None):
    print(root_folder.title)
```

or you can leave the arguments empty:

```
for root_folder in client.descendants():
    print(root_folder.title)
```

The `descendants` method is a generator function. The method behaves like an iterator, i.e. it can be used in a for loop, the advantage of this approach is that the paging of results is taken care of automatically. If a Folder has many thousands of Assets then the method will make multiple calls to the server. It will default to 100 items between server requests.

The performance improvement from the use of generators is the result of the lazy (on demand) generation of values, which translates to lower memory usage. Furthermore, you do not need to wait until all the children have been generated

before you start to use them.

To get a set of the immediate children of a particular Folder use

```
for entity in client.descendants(folder.reference):
    print(entity.title)
```

To get the siblings of an Asset you can use

```
for entity in client.descendants(asset.parent):
    print(entity.title)
```

The set of entities returned may contain both Assets and other Folders.

Note: Entities within the returned set only contain the attributes (type, reference and title). If you need the full object you have to request it from the server.

You can request the entity back without knowing exactly what type it is by using the `entity()` call

To fetch the full object back you can use:

```
for f in client.descendants():
    e = client.entity(f.entity_type, f.reference)
    print(e)
```

If you only need the Folders or Assets from a parent you can filter the results using a pre-defined filter.

For example the following will only return Asset objects and will ignore Folders:

```
for asset in filter(only_assets, client.descendants(asset.parent)):
    print(asset.title)
```

To return only Folder objects use:

```
for folders in filter(only_folders, client.descendants(asset.parent)):
    print(folders.title)
```

If you want **all** the entities below a point in the hierarchy, i.e a recursive list of all folders and Assets then you can call `all_descendants()` this is also generator function which returns a lazy iterator which will make repeated calls to the server for each page of results.

The following will return all entities within the repository from the root folders down

```
for e in client.all_descendants():
    print(e.title)
```

Warning: The code above will fetch every Asset or Folder back from the system. This could take a long time depending on the size of the repository.

It may be more efficient to search using the ContentAPI if you are looking for particular objects in the repository.

again if you need a list of every Asset in the system you can filter using

```
for asset in filter(only_assets, client.all_descendants()):
    print(asset.title)
```

13.3 Creating new Folders

Folder objects can be created directly in the repository, the `create_folder()` function takes 3 mandatory parameters, folder title, description and security tag.

```
new_folder = client.create_folder("title", "description", "open")
print(new_folder.reference)
```

This will create a folder at the top level of the repository. You can create child folders by passing the reference of the parent as the last argument.

```
new_folder = client.create_folder("title", "description", "open", folder.reference)
print(new_folder.reference)
assert new_folder.parent == folder.reference
```

13.4 Adding Physical Assets

Preservica supports the creation of intellectual entities which correspond to physical objects. These are similar to regular assets, but they do not point to digital files like regular assets.

To use Physical Assets the system needs a system property set to active the functionality, this can be done by the Preservica help desk.

```
parent = client.folder("9bad5acf-e7a1-458a-927d-2d1e7f15974d")
physical_asset = client.add_physical_asset("title", "description", parent, "open")
print(physical_asset.reference)
```

Physical assets support 3rd party identifiers, thumbnails and descriptive metadata in the same way as regular assets.

```
client.add_identifier(physical_asset, "ISBN", "978-3-16-148410-0")
client.add_thumbnail(physical_asset, "icon.png")
```

13.5 Updating Entities

We can update either the title or description attribute for Assets, Folders and Content Objects using the `save()` method

```
asset = client.asset("9bad5acf-e7a1-458a-927d-2d1e7f15974d")
asset.title = "New Asset Title"
asset.description = "New Asset Description"
asset = client.save(asset)

folder = client.folder("0b0f0303-6053-4d4e-a638-4f6b81768264")
folder.title = "New Folder Title"
folder.description = "New Folder Description"
folder = client.save(folder)

content_object = client.content_object("1a2a2101-6053-4d4e-a638-4f6b81768264")
content_object.title = "New Content Object Title"
content_object.description = "New Content Object Description"
content_object = client.save(content_object)
```


This method can also be used to set the Type of an asset or folder. By default Information objects have a type “Asset” and Structural objects have a type “Folder”. You can use the API to change these defaults for example you may want to use the type field to set the level of description of a Structural object to “Fonds” or “Series” etc.

To change the type use the *custom_type* attribute on the object, e.g.

```
folder = client.folder("9bad5acf-e7a1-458a-927d-2d1e7f15974d")
folder.custom_type = "Series"
folder = client.save(folder)
```

```
asset = client.asset("9bad5acf-e7a1-458a-927d-2d1e7f15974d")
asset.custom_type = "Manuscript"
asset = client.save(asset)
```

If you want to change the type back, just set the value to None

```
asset = client.asset("9bad5acf-e7a1-458a-927d-2d1e7f15974d")
asset.custom_type = None
asset = client.save(asset)
```

13.6 Security Tags

To change the security tag on an Asset or Folder we have a separate API. Since this may be a long running process. You can choose either a asynchronous (non-blocking) call which returns immediately or synchronous (blocking call) which waits for the security tag to be changed before returning.

This is the asynchronous call which returns immediately returning a process id

```
pid = client.security_tag_async(entity, new_tag)
```

You can determine the current status of the asynchronous call by passing the argument to *get_async_progress*

```
status = client.get_async_progress(pid)
```

The synchronous version will block until the security tag has been updated on the entity. This call does not recursively change entities within a folder. It only applies to the named entity passed as an argument.

```
entity = client.security_tag_sync(entity, new_tag)
```

13.7 3rd Party External Identifiers

3rd party or external identifiers are a useful way to provide additional names or identities to objects to provide an alternate way of accessing them. For example if you are synchronising metadata between an external metadata catalogue and Preservica adding the catalogue identifiers to the Preservica objects allows the catalogue to query Preservica using its own ids.

Each Preservica entity can hold as many external identifiers as you need.

Note: Adding, Updating and Deleting external identifiers is only available in version 6.1 and above

We can add external identifiers to either Assets, Folders or Content Objects. External identifiers have a name or type and a value. External identifiers do not have to be unique in the same way as internal identifiers. The same external identifiers can be added to multiple entities to form sets of objects.

```
asset = client.asset("9bad5acf-e7ce-458a-927d-2d1e7f15974d")
client.add_identifier(asset, "ISBN", "978-3-16-148410-0")
client.add_identifier(asset, "DOI", "https://doi.org/10.1109/5.771073")
client.add_identifier(asset, "URN", "urn:isan:0000-0000-2CEA-0000-1-0000-0000-Y")
```

Fetch external identifiers on an entity. This call returns a set of tuples (identifier_type, identifier_value)

```
identifiers = client.identifiers_for_entity(folder)
for identifier in identifiers:
    identifier_type = identifier[0]
    identifier_value = identifier[1]
```

You can search the repository for entities with matching external identifiers. The call returns a set of objects which may include any type of entity.

```
for e in client.identifier("ISBN", "978-3-16-148410-0"):
    print(e.entity_type, e.reference, e.title)
```

Note: Entities within the set only contain the attributes (type, reference and title). If you need the full object you have to request it.

For example

```
for ident in client.identifier("DOI", "urn:nbn:de:1111-20091210269"):
    entity = client.entity(ident.entity_type, ident.reference)
    print(entity.title)
    print(entity.description)
```

To delete identifiers attached to an entity

```
client.delete_identifiers(entity)
```

Will delete all identifiers on the entity

```
client.delete_identifiers(entity, identifier_type="ISBN")
```

Will delete all identifiers which have type “ISBN”

```
client.delete_identifiers(entity, identifier_type="ISBN", identifier_value="978-3-16-
↪148410-0")
```

Will only delete identifiers which match the type and value

13.8 Descriptive Metadata

You can query an entity to determine if it has any attached descriptive metadata using the metadata attribute. This returns a dictionary object the dictionary key is a url which can be used to the fetch metadata and the value is the schema name

```
for url, schema in entity.metadata.items():
    print(url, schema)
```

The descriptive XML metadata document can be returned as a string by passing the key of the map (url) to the metadata() method

```
for url in entity.metadata:
    xml_string = client.metadata(url)
```

An alternative is to call the metadata_for_entity directly

```
xml_string = client.metadata_for_entity(entity, "https://person.org/person")
```

this will fetch the first metadata document which matches the schema argument on the entity

If you need all the descriptive XML fragments attached to an Asset or Folder you can call all_metadata this is a Generator which returns a Tuple containing the schema as the first item and the xml document in the second.

```
for metadata in client.all_metadata(entity):
    schema = metadata[0]
    xml_string = metadata[1]
```

Metadata can be attached to entities either by passing an XML document as a string

```
folder = entity.folder("723f6f27-c894-4ce0-8e58-4c15a526330e")

xml = "<person:Person xmlns:person='https://person.org/person'>" \
    "<person:Name>Bob Smith</person:Name>" \
    "<person:Phone>01234 100 100</person:Phone>" \
    "<person:Email>test@test.com</person:Email>" \
    "<person:Address>Abingdon, UK</person:Address>" \
    "</person:Person>"

folder = client.add_metadata(folder, "https://person.org/person", xml)
```

or by reading the metadata from a file

```
with open("DublinCore.xml", 'r', encoding="utf-8") as md:
    asset = client.add_metadata(asset, "http://purl.org/dc/elements/1.1/", md)
```

Adding descriptive metadata may change the namespace prefix values, this does not change the meaning of the XML document as the prefix values are arbitrary labels. XML namespace prefixes themselves are arbitrary; it's only through their binding to a full XML namespace name that they derive their significance.

If you want to preserve the namespace prefix you can add the following to the start of your Python scripts

```
xml.etree.ElementTree.register_namespace("person", "https://person.org/person")
```

This will associate the namespace prefix “person” with the actual XML namespace

Descriptive metadata can also be updated to amend values or change the document structure. To update an existing metadata document call

```
client.update_metadata(entity, schema, xml_string)
```

For example the following python fragment appends a new element to an existing document.

```
folder = client.folder("723f6f27-c894-4ce0-8e58-4c15a526330e") # call into the API

for url, schema in folder.metadata.items():
    if schema == "https://person.org/person":
        xml_string = client.metadata(url) # call into the API
        xml_document = ElementTree.fromstring(xml_string)
        postcode = ElementTree.Element('{https://person.org/person}Postcode')
        postcode.text = "OX14 3YS"
        xml_document.append(postcode)
        xml_string = ElementTree.tostring(xml_document, encoding='UTF-8').decode("utf-8")
        client.update_metadata(folder, schema, xml_string) # call into the API
```

13.9 Relationships Between Entities

Preservica allows arbitrary relationships between entities such as Assets and Folders. These relationships appear in the Preservica user interface as links from one entity to another. All entities have existing vertical parent child relationships which determine the level of description for an asset. These relationships are additional relationships which relate different entities across the repository.

For example relationships may be used to link different editions of the same work, or a translation of an existing document etc.

Any type of relationship is supported, for example The Dublin Core Metadata Initiative provide a set of standard relationships between entities, and these have been provided as part of the Relationship class, but any text string is allowed for the relationship type.

```
>>>Relationship.DCMI_isVersionOf
http://purl.org/dc/terms/isVersionOf

>>>Relationship.DCMI_isReplacedBy
http://purl.org/dc/terms/isReplacedBy
```

Relationships are created between two entities A and B and have a type, for example;

A isVersionOf B.

This is a relationship from A to B. You can also create links going in the other direction and have bi-directional links between the same assets. For example;

A isVersionOf B and B hasVersion A.

To create a relationship between entities use the `add_relation` method.

```
A_asset = client.asset("de1c32a3-bd9f-4843-a5f1-46df080f83d2")
B_asset = client.asset("683f9db7-ff81-4859-9c03-f68cfa5d9c3d")

client.add_relation(A_asset, Relationship.DCMI_isVersionOf, B_asset)
client.add_relation(B_asset, Relationship.DCMI_hasVersion, A_asset)
```

(continues on next page)

(continued from previous page)

```
client.add_relation(A_asset, "Supersedes", B_asset)
```

Note: The Relationship API is only available when connected to Preservica version 6.3.1 or above

You can list the relationships from an asset using:

```
for r in client.relationships(A_asset):
    print(r)
```

This returns a Generator of Relationship objects.

To delete relationships between assets use:

```
client.delete_relationships(A_asset)
```

This will delete all relationships FROM the specified entity to another entity, It does not delete relationships TO this entity.

If only need to delete a specific relationship, you can pass the relationship name as a second argument

```
client.delete_relationships(A_asset, "Supersedes")
```

13.10 Representations, Content Objects & Generations

Each Asset in Preservica contains one or more representations, such as Preservation or Access etc. All Assets have at least one Preservation representation which is created when the Asset is ingested.

To get a list of all the representations of an Asset use `representations()` which returns a set of `Representation` objects for the Asset.

The `Representation` contains the name and type and also contains a reference back to its parent Asset object.

Currently Preservica supports two representation types “Access” and “Preservation”, you can have as many representations of each type as you need. For example a book may need two “Access” representations one containing a single PDF document and another containing multiple JPEG images, one for each page etc.

```
for representation in client.representations(asset):
    print(representation.rep_type)
    print(representation.name)
    print(representation.asset.title)
```

Each Representation will contain one or more Content Objects. Simple Assets contain a single Content Object per Representation whereas more complex objects such as 3D models, books, multi-page documents may have several content objects within each Representation.

Content Objects are similar to Assets and Folders, in that they can also contain descriptive metadata and identifiers etc. The Content Objects within a Representation do have a natural order which is preserved within the Asset and therefore are returned as a list object.

```
for content_object in client.content_objects(representation):
    print(content_object.reference)
```

(continues on next page)

(continued from previous page)

```
print(content_object.title)
print(content_object.description)
print(content_object.parent)
print(content_object.metadata)
print(content_object.asset.title)
```

By default the title of a Content Object will probably be the name of the underlying computer file, but it does not have to be. You can explicitly set the title and description of each Content Object within an Asset. Preservica also supports adding external identifiers and descriptive metadata documents to Content Objects.

Each Content Object will contain a least one Generation, migrated content may have multiple Generations.

```
for generation in client.generations(content_object):
    print(generation.original)
    print(generation.active)
    print(generation.content_object)
    print(generation.format_group)
    print(generation.effective_date)
    print(generation.bitstreams)
```

Each Generation has a list of BitStreams which can be used to fetch the actual content from the server or fetch technical metadata about the bitstream itself.

Technical information such as formats and properties can be accessed from the `Generation` object. The format information is stored as dictionary object within a list as there may be multiple formats associated with each object.

The key values for the format dictionary are: Valid, PUID, Priority, IdentificationMethod, FormatName, FormatVersion

```
for format in generation.formats:
    for key,value in format.items():
        print(key, value)
```

The technical properties of the file can be accessed via the `properties` attribute which is a list of dictionary objects. Each property is a single dictionary object with the following keys: PUID, PropertyName, Value

```
for property in generation.properties:
    for key,value in property.items():
        print(key, value)
```

Generations also contain a list of bitstreams, these contain information about the bitstreams such as file size and fixity etc.

```
for bitstream in generation.bitstreams:
    print(bitstream.filename)
    print(bitstream.length)
    for algorithm,value in bitstream.fixity.items():
        print(algorithm, value)
```

If you have an Asset object and you would like to fetch all the available bitstreams you would use something like:

```
for representation in client.representations(asset):
    for content_object in client.content_objects(representation):
        for generation in client.generations(content_object):
            for bitstream in generation.bitstreams:
```

If you only need the current or active Generations, then you can use the following short cut method which returns each Bitstream from all the Representations and Content Objects within the Asset.

```
for bitstream in client.bitstreams_for_asset(asset):
    do_something(bitstream)
```

The actual content files can be downloaded to a disk file using `bitstream_content()`

This will download the bitstream to the file path given by the second argument, to save the object using the original file name use the following:

```
client.bitstream_content(bitstream, bitstream.filename)
```

To download all the access bitstreams to the current folder you would use.

```
for representation in client.representations(asset):
    if representation.rep_type == "Access":
        for content_object in client.content_objects(representation):
            for generation in client.generations(content_object):
                for bitstream in generation.bitstreams:
                    client.bitstream_content(bitstream, bitstream.filename)
```

The content files can be written to a byte array using `bitstream_bytes()` this returns a BytesIO object.

```
byte_array = client.bitstream_bytes(bitstream)
```

If you need to process bitstream content as it is downloaded from Preservica pyPreservica provides the following API.

```
for bitstream in client.bitstreams_for_asset(asset):
    for chunk in client.bitstream_chunks(bitstream):
        doSomething(chunk)
```

This function returns a Generator which allows the client to process parts of the file as its downloading.

The method also allows a second argument which defines the size of chunk returned.

```
chunk_size8k = 8*1024
for bitstream in client.bitstreams_for_asset(asset):
    for chunk in client.bitstream_chunks(bitstream, chunk_size8k):
        doSomething(chunk)
```

Since version Preservica 6.12 the API allows new Access representations to be added to an existing Asset. This allows organisations to migrate content outside of Preservica or add new access versions after the preservation versions have been ingested.

To add a new Access representation to an existing Asset call `add_access_representation` and pass the Asset and a new content file. The function returns a process id which can be used to track the status of the ingest.

The Preservica tenancy requires the `post.new.representation.feature` flag to be set.

```
asset = client.asset("723f6f27-c894-4ce0-8e58-4c15a526330e")
pid = client.add_access_representation(asset, access_file="access.jpg")
```

13.11 Integrity Check History

You can request the history of all integrity checks which have been carried out on a bitstream

```
for bitstream in generation.bitstreams:
    for check in client.integrity_checks(bitstream):
        print(check)
```

The list of returned checks includes both full and quick integrity checks.

Note: This call does not start a new check, it only returns information about previous checks.

13.12 Moving Entities

We can move entities between folders using the `move` call

```
client.move(entity, dest_folder)
```

Where `entity` is the object to move either an `Asset` or `Folder` and the second argument is destination folder where the entity is moved to.

Folders can be moved to the root of the repository by passing `None` as the second argument.

```
entity = client.move(folder, None)
```

The `move()` call is an alias for `move_sync()` which is a synchronous (blocking call)

```
entity = client.move_sync(entity, dest_folder)
```

An asynchronous (non-blocking) version is also available which returns a progress id.

```
pid = client.move_async(entity, dest_folder)
```

You can determine the completed status of the asynchronous move call by passing the argument to `get_async_progress`

```
status = client.get_async_progress(pid)
```

13.13 Deleting Entities

You can initiate and approve a deletion request using the API.

Note: Deletion is a two stage process within Preservica and requires two distinct sets of credentials. To use the delete functions you must be using the “credentials.properties” authentication method.

Note: The Deletion API is only available when connected to Preservica version 6.2 or above

Add `manager.username` and `manager.password` to the credentials file.


```
[credentials]
username=
password=
server=
tenant=
manager.username=
manager.password=
```

Deleting an asset

```
asset_ref = client.delete_asset(asset, "operator comments", "supervisor comments")
print(asset_ref)
```

Deleting a folder

```
folder_ref = client.delete_folder(folder, "operator comments", "supervisor comments")
print(folder_ref)
```

Warning: This API call deletes entities within the repository, it both initiates and approves the deletion request and therefore must be used with care.

13.14 Finding Updated Entities

We can query Preservica for entities which have changed over the last n days using

```
for e in client.updated_entities(previous_days=30):
    print(e)
```

The argument is the number of previous days to check for changes. This call does paging internally.

13.15 Downloading Files

The pyPreservica library also provides a web service call which is part of the content API which allows downloading of digital content directly without having to request the Representations and Generations first. This call is a short-cut to request the Bitstream from the latest Generation of the first Content Object in the Access Representation of an Asset. If the asset does not have an Access Representation then the Preservation Representation is used.

For very simple assets which comprise a single digital file in a single Representation then this call will probably do what you expect.

```
asset = client.asset("edf403d0-04af-46b0-ab21-e7a620bfdedf")
filename = client.download(asset, "asset.jpg")
```

For complex multi-part assets which have been through preservation actions it may be better to use the data model and the `bitstream_content()` function to fetch the exact bitstream you need.

13.16 Events on Specific Entities

List actions performed against this entity

`entity_events()` returns a iterator which contains events on an entity, either an asset or folder

```
asset = client.asset("edf403d0-04af-46b0-ab21-e7a620bfdedf")
for event in client.entity_events(asset):
    print(event)
```

13.17 Events Across Entities

List actions performed against all entities within the repository. The event is a `dict()` object containing the event attributes. This call is generator function which returns the events as needed.

```
for event in client.all_events():
    print(event)
```

13.18 Ingest Events

Return a generator of ingest events over the last n days

```
for ingest_event in client.all_ingest_events(previous_days=1):
    print(ingest_event)
```

13.19 Asset and Folder Thumbnail Images

You can now add and remove icons on Assets and Folders using the API. The icons will be displayed in the Explorer and Universal Access interfaces.

```
folder = client.folder("edf403d0-04af-46b0-ab21-e7a620bfdedf")
client.add_thumbnail(folder, "../my-icon.png")

client.remove_thumbnail(folder)
```

and for assets

```
asset = client.asset("edf403d0-04af-46b0-ab21-e7a620bfdedf")
client.add_thumbnail(asset, "../my-icon.png")

client.remove_thumbnail(asset)
```

We also have a function to fetch the thumbnail image for an asset or folder

```
asset = client.asset("edf403d0-04af-46b0-ab21-e7a620bfdedf")
filename = client.thumbnail(asset, "thumbnail.png")
```

You can specify the size of the thumbnail by passing a second argument

```
asset = client.asset("edf403d0-04af-46b0-ab21-e7a620bfdedf")
filename = client.thumbnail(asset, "thumbnail.png", Thumbnail.LARGE)    ## 400x400  ↵
↪pixels
filename = client.thumbnail(asset, "thumbnail.png", Thumbnail.MEDIUM)  ## 150x150  ↵
↪pixels
filename = client.thumbnail(asset, "thumbnail.png", Thumbnail.SMALL)    ## 64x64    ↵
↪pixels
```

13.20 Replacing Content Objects

Preservica now supports replacing individual Content Objects within an Asset. The use case here is you have uploaded a large digitised object such as book and you subsequently discover that a page has been digitised incorrectly. You would like to replace a single page (Content Object) without having to delete and re-ingest the complete Asset.

The non-blocking (asynchronous) API call will replace the last active Generation of the Content Object

```
content_object = client.content_object('0f2997f7-728c-4e55-9f92-381ed1260d70')
file = "C:/book/page421.tiff"
pid = client.replace_generation_async(content_object, file)
```

This will return a process id which can be used to monitor the replacement workflow using

```
status = client.get_async_progress(pid)
```

By default the API will generate a new fixity value on the client using the same fixity algorithm as the original Generation you are replacing. If you want to use a different fixity algorithm or you want to use a pre-calculated or existing fixity value you can specify the algorithm and value.

```
content_object = client.content_object('0f2997f7-728c-4e55-9f92-381ed1260d70')
file = "C:/book/page421.tiff"
pid = client.replace_generation_async(content_object, file, fixity_algorithm='SHA1', ↵
↪fixity_value='2fd4e1c67a2d28fced849ee1bb76e7391b93eb12')
```

There is also an synchronous or blocking version which will wait for the replace workflow to complete before returning back to the caller.

```
content_object = client.content_object('0f2997f7-728c-4e55-9f92-381ed1260d70')
file = "C:/book/page421.tiff"
workflow_status = client.replace_generation_sync(content_object, file)
```

13.21 Export OPEX Package

pyPreservica allows clients to request a full package export from the system by folder or asset, this will start an export workflow and download the resulting dissemination package when the export workflow has completed.

The resulting package will be a zipped OPEX formatted package containing the digital content and metadata. The `export_opex` API is a blocking call which will wait for the export workflow to complete before downloading the package.

```
folder = client.folder('0f2997f7-728c-4e55-9f92-381ed1260d70')
opex_zip = client.export_opex(folder)
```

The output is the name of the downloaded zip file in the current working directory.

By default the OPEX package includes metadata, digital content with the latest active generations and the parent hierarchy.

The API can be called on either a folder or a single asset.

```
asset = client.asset('1f2129f7-728c-4e55-9f92-381ed1260d70')
opex_zip = client.export_opex(asset)
```

The call also takes the following optional arguments

- IncludeContent “Content” or “NoContent”
- IncludeMetadata “Metadata” or “NoMetadata” or “MetadataWithEvents”
- IncludedGenerations “LatestActive” or “AllActive” or “All”
- IncludeParentHierarchy “true” or “false”

e.g.

```
folder = client.folder('0f2997f7-728c-4e55-9f92-381ed1260d70')
opex_zip = client.export_opex(folder, IncludeContent="Content", IncludeMetadata=
↳ "MetadataWithEvents")
```

CONTENT API

pyPreservica now contains interfaces to the content API which supports searching the repository.

<https://us.preservica.com/api/content/documentation.html>

The content API is a readonly interface which returns json documents rather than XML and which has some duplication with the entity API, but it does contain search capabilities.

The content API client is created using

```
from pyPreservica import *  
  
client = ContentAPI()
```

14.1 object-details

Get the details for a Asset or Folder as a Python dictionary object containing CMIS attributes

```
client = ContentAPI()  
  
client.object_details("IO", "uuid")  
client.object_details("SO", "uuid")
```

e.g.

```
from pyPreservica import *  
  
client = ContentAPI()  
  
details = client.object_details("IO", "de1c32a3-bd9f-4843-a5f1-46df080f83d2")  
print(details['name'])
```

or

```
from pyPreservica import *  
  
client = ContentAPI()  
  
details = client.object_details(EntityType.ASSET, "de1c32a3-bd9f-4843-a5f1-46df080f83d2")  
print(details['name'])
```

14.2 indexed-fields

Get a list of all the indexed metadata fields within the Preservica search engine. This includes the default xip.* fields and any custom indexes which have been created through custom index files.

```
client = ContentAPI()

client.indexed_fields():
```

14.3 Search

Search the repository using a single expression which matches on any indexed field.

```
client = ContentAPI()

client.simple_search_csv()
```

Searches for everything and writes the results to a csv file called “search.csv”, by default the csv columns contain reference, title, description, document_type, parent_ref, security_tag.

You can pass the query term as the first argument (% is the wildcard character) and the csv file name as the second argument.

```
client = ContentAPI()

client.simple_search_csv("%", "everything.csv")

client.simple_search_csv("Oxford", "oxford.csv")

client.simple_search_csv("History of Oxford", "history.csv")
```

The last argument is an optional list of indexed fields which are the csv file columns.

```
client = ContentAPI()

metadata_fields = ["xip.reference", "xip.title", "xip.description", "xip.document_type",
↳ "xip.parent_ref", "xip.security_descriptor"]
client.simple_search_csv("%", "results.csv", metadata_fields)
```

or to include everything except the full text index value

```
client = ContentAPI()

everything = list(filter(lambda x: x != "xip.full_text", client.indexed_fields()))
client.simple_search_csv("%", "results.csv", everything)
```

There is an equivalent call which does not write the output to CSV, but returns a generator of dictionary objects. This is useful if you want to process the results within the script and not generate a report directly.

```
client = ContentAPI()

for hit in client.simple_search_list("History of Oxford"):
    print(hit)
```

and

```
client = ContentAPI()

metadata_fields = ["xip.reference", "xip.title", "xip.description", "xip.document_type",
↳ "xip.parent_ref", "xip.security_descriptor"]
for hit in client.simple_search_list("History of Oxford", metadata_fields):
    print(hit['xip.title'])
```

If you want to do searches with advanced filter terms then the following calls can be used. These calls use a Python dictionary to allow the caller to specify filter values on the indexed terms.

```
client = ContentAPI()

filters = {"dc.rights": "Public Domain", "xip.security_descriptor": "public"}
for hit in client.search_index_filter_list(query="History of Oxford", filter_
↳ values=filters):
    print(hit)
```

If you want to generate a report which can be opened directly in Excel, the use the csv version.

```
client = ContentAPI()

filters = {"oai_dc.contributor": "*", "xip.security_descriptor": "public"}
client.search_index_filter_csv(query="History of Oxford", csv_file="my-report.csv",
↳ filter_values=filters)
```

The special filter value "*" is used to filter indexes which have a value, i.e. are values are not empty or missing. The filter value "%" is used to specify any value including empty values.

For example to create a report on the security tags of all assets within a folder you can use

```
client = ContentAPI()

filters = {"xip.title": "%", "xip.description": "%", "xip.security_descriptor": "*",
↳ "xip.parent_ref": "48c79abd-01f3-4b77-8132-546a76e0d337"}
client.search_index_filter_csv(query="%", csv_file="security.csv", filter_values=filters)
```

14.4 Search Progress

Searching across a large Preservica repository is very quick, but returning very large datasets back to the client can be slow. To avoid putting undue load on the server pyPreservica will request a single page of results at a time for each server request.

If you are using the `simple_search_csv` or `search_index_filter_csv` functions which write directly to a csv file then it can be difficult to monitor the report generation progress.

To allow allow monitoring of search result downloads, you can add a callback to the search client. The callback class will be called for every page of search results returned to the client. The value passed to the callback contains the total number of search hits for the query and the current number of results processed.

Preservica provides a default callback

```

class ReportProgressCallBack:
    def __init__(self):
        self.current = 0
        self.total = 0
        self._lock = threading.Lock()

    def __call__(self, value):
        with self._lock:
            values = value.split(":")
            self.total = int(values[1])
            self.current = int(values[0])
            percentage = (self.current / self.total) * 100
            sys.stdout.write("\r%s / %s (%.2f%%)" % (self.current, self.total,
↪percentage))
            sys.stdout.flush()

```

To use the default callback in your scripts include the following line

```
client.search_callback(client.ReportProgressCallBack())
```

14.5 Reporting Examples

14.5.1 Create a spreadsheet containing all Assets within the repository

Generate a CSV report on all assets within the system, spreadsheet columns include asset title, description, security tag etc

```

from pyPreservica import *

client = ContentAPI()

if __name__ == '__main__':
    metadata_fields = {
        "xip.reference": "*", "xip.title": "", "xip.description": "", "xip.document_type
↪": "IO", "xip.parent_ref": "",
        "xip.security_descriptor": "*",
        "xip.identifier": "", "xip.bitstream_names_r_Preservation": ""}

    client.search_callback(client.ReportProgressCallBack())

    client.search_index_filter_csv("", "assets.csv", metadata_fields)

```


14.5.2 Create a spreadsheet containing all Assets and Folders within the repository

```
from pyPreservica import *

client = ContentAPI()

if __name__ == '__main__':
    metadata_fields = {
        "xip.reference": "*", "xip.title": "", "xip.description": "", "xip.document_type": "",
        "xip.parent_ref": "",
        "xip.security_descriptor": "*",
        "xip.identifier": "", "xip.bitstream_names_r_Preservation": ""}

    client.search_callback(client.ReportProgressCallBack())

    client.search_index_filter_csv("%", "all_objects.csv", metadata_fields)
```

14.5.3 Create a spreadsheet containing all Assets and Folders underneath a specific folder

```
from pyPreservica import *

content = ContentAPI()
entity = EntityAPI()

folder = entity.folder(sys.argv[1])

print(f"Searching inside folder {folder.title}")

if __name__ == '__main__':
    metadata_fields = {
        "xip.reference": "*", "xip.title": "", "xip.description": "", "xip.document_type": "",
        "xip.parent_hierarchy": f"{folder.reference}",
        "xip.security_descriptor": "*",
        "xip.identifier": "", "xip.bitstream_names_r_Preservation": ""}

    content.search_callback(content.ReportProgressCallBack())

    content.search_index_filter_csv("%", "assets.csv", metadata_fields)
```

14.6 User Security Tags

You can get a list of available security tags for the current user by calling:

```
client = ContentAPI()
client.user_security_tags()
```

UPLOAD API

PyPreservica provides some limited capabilities for the Upload Content API

<https://developers.preservica.com/api-reference/3-upload-content-s3-compatible>

The Upload API can be used for creating, uploading and automatically starting an ingest workflows with pre-created packages. The Package can be either a native v5 SIP as created from a tool such as the SIP Creator or a native v6 SIP created manually. Zipped OPEX packages are also supported. <https://developers.preservica.com/documentation/open-preservation-exchange-opex>

The package can also be a regular zip file containing just folders and files with or without simple .metadata files.

15.1 Uploading Packages

The upload API client is created using

```
from pyPreservica import *  
  
upload = UploadAPI()
```

Once you have a client you can use it to upload packages.

```
upload.upload_zip_package("my-package.zip")
```

Will upload the local zip file and start an ingest workflow if one is enabled.

The zip file can be any of the following:

- Zipped Native XIPv5 Package (i.e. created from the SIP Creator)
- Zipped Native XIPv6 Package (see below)
- Zipped OPEX Package
- Zipped Folder

Note: A Workflow Context must be active for the package upload requests to be successful.

If the package is a simple zipped folder without a manifest XML then you will want to pass information to the ingest to specify which folder the content should be ingested into. To specify the parent folder of the ingest pass a folder object as the second argument.

```
upload = UploadAPI()
client = EntityAPI()

folder = client.folder("edf403d0-04af-46b0-ab21-e7a620bfdedf")
upload.upload_zip_package(path_to_zip_package="my-package.zip", folder=folder)
```

For large packages it is more reliable to send the submission via the AWS S3 transfer bucket connected to a ingest workflow. The available transfer buckets are shown on the Preservica administration sources tab. The ingest can then be triggered automatically once the submission is saved to the S3 transfer bucket.

```
upload = UploadAPI()
client = EntityAPI()

folder = client.folder("edf403d0-04af-46b0-ab21-e7a620bfdedf")
bucket = "com.preservica.<Tenant-ID>.upload"
upload.upload_zip_package_to_S3(path_to_zip_package="my-large-package.zip", bucket_
↳name=bucket, folder=folder)
```

Note: This upload method is only available to AWS users.

If your Preservica system is deployed on Azure you can use:

```
upload = UploadAPI()
client = EntityAPI()

folder = client.folder("edf403d0-04af-46b0-ab21-e7a620bfdedf")
bucket = "com.preservica.<Tenant-ID>.upload"
upload.upload_zip_package_to_Azure(path_to_zip_package="my-large-package.zip", container_
↳name=bucket, folder=folder)
```

If you are writing client code which could be used on both AWS or Azure platforms than you can use the following which will upload into a monitored cloud location on either platform

```
upload = UploadAPI()
client = EntityAPI()

folder = client.folder("edf403d0-04af-46b0-ab21-e7a620bfdedf")
bucket = "com.preservica.<Tenant-ID>.upload"
upload.upload_zip_to_Source(path_to_zip_package="my-large-package.zip", container_
↳name=bucket, folder=folder)
```

15.2 Monitoring Upload Progress

The `upload_zip_package` function accepts an optional `Callback` parameter. The parameter references a class that pyPreservica invokes intermittently during the transfer operation.

pyPreservica executes the class's `__call__` method. For each invocation, the class is passed the number of bytes transferred up to that point. This information can be used to implement a progress monitor.

The following `Callback` setting instructs pyPreservica to create an instance of the `UploadProgressCallback` class. During the upload, the instance's `__call__` method will be invoked intermittently.

```
from pyPreservica import UploadProgressCallback
my_callback=UploadProgressCallback("my-package.zip")
client.upload_zip_package(path_to_zip_package="my-package.zip", folder=folder,
↳callback=my_callback)
```

The default pyPreservica UploadProgressCallback looks like

```
import os
import sys
import threading

class ProgressPercentage(object):
    def __init__(self, filename):
        self._filename = filename
        self._size = float(os.path.getsize(filename))
        self._seen_so_far = 0
        self._lock = threading.Lock()

    def __call__(self, bytes_amount):
        with self._lock:
            self._seen_so_far += bytes_amount
            percentage = (self._seen_so_far / self._size) * 100
            sys.stdout.write("\r%s %s / %s (%.2f%%)" % (self._filename, self._seen_so_
↳far, self._size, percentage))
            sys.stdout.flush()
```

15.3 Creating Packages

The UploadAPI module also contains functions for creating XIPv6 packages directly from content files.

To create a package containing a single preservation Content Object (file) as part of an Asset which will be a child of specified folder

```
package_path = simple_asset_package(preservation_file="my-image.tiff", parent_
↳folder=folder)
```

The output is a path to the zip file which can be passed directly to the upload_zip_package method

```
client.upload_zip_package(path_to_zip_package=package_path)
```

By default the Asset title and description will be taken from the file name.

If you don't specify an export folder the new package will be created in the system TEMP folder. If you want to override this behaviour and explicitly specify the output folder for the package use the export_folder argument

```
package_path = simple_asset_package(preservation_file="my-image.tiff", parent_
↳folder=folder,
                                export_folder="/mnt/export/packages")
```

You can specify the Asset title and description using additional keyword arguments.

```
package_path = simple_asset_package(preservation_file="my-image.tiff", parent_
↳ folder=folder,
                                     Title="Asset Title", Description="Asset Description")
```

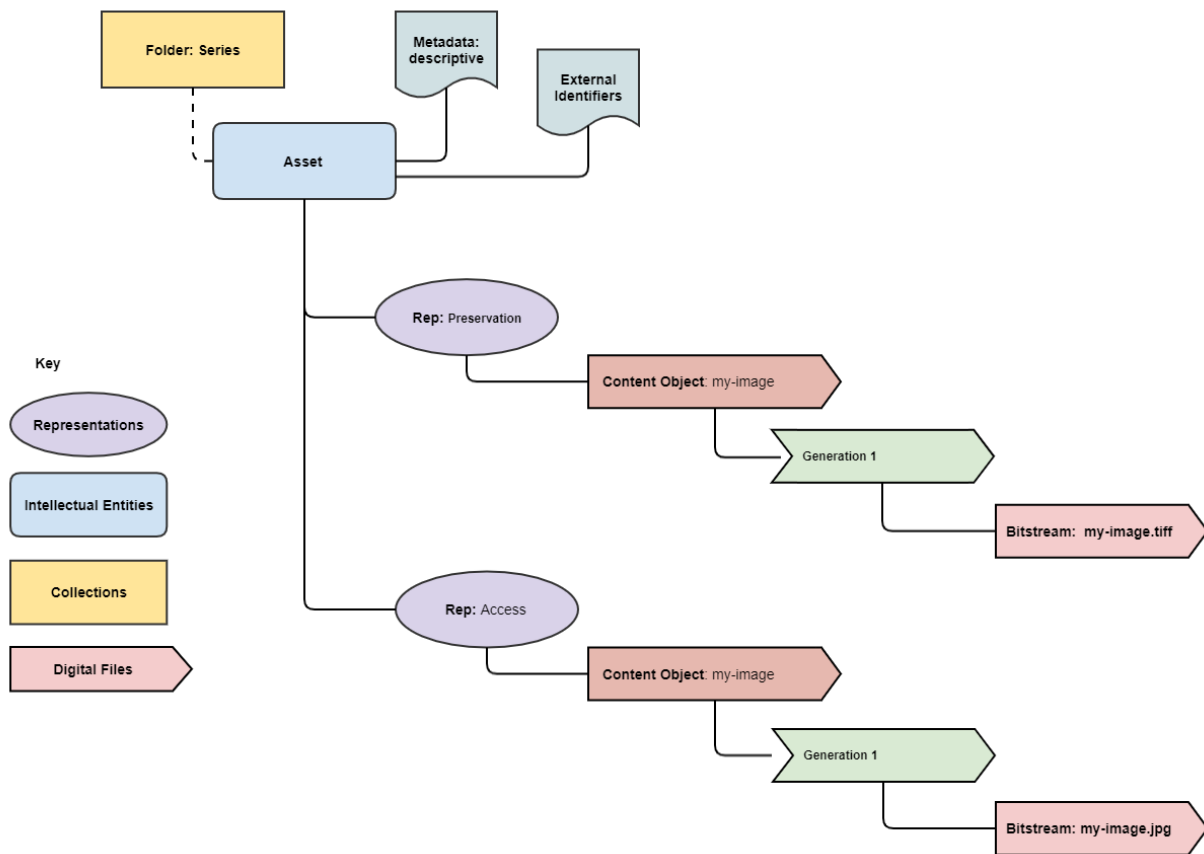
You can also add a second Access content object to the asset. This will create an asset with two representations (Preservation & Access)

```
package_path = simple_asset_package(preservation_file="my-image.tiff", access_file="my-
↳ image.jpg"
                                     parent_folder=folder)
```

It is possible to configure the asset within the package using the following additional keyword arguments.

- **Title** Asset Title
- **Description** Asset Description
- **SecurityTag** Asset Security Tag
- **CustomType** Asset Type
- **Preservation_Content_Title** Content Object Title of the Preservation Object
- **Preservation_Content_Description** Content Object Description of the Preservation Object
- **Access_Content_Title** Content Object Title of the Access Object
- **Access_Content_Description** Content Object Description of the Access Object
- **Preservation_Generation_Label** Generation Label for the Preservation Object
- **Access_Generation_Label** Generation Label for the Access Object
- **Asset_Metadata** Dictionary of metadata schema/documents to add to the Asset
- **Identifiers** Dictionary of Asset identifiers
- **Preservation_files_fixity_callback** Fixity generation callback for preservation files
- **Access_files_fixity_callback** Fixity generation callback for access files

The package will contain an asset with the following structure.



For example to add descriptive metadata and two 3rd party identifiers use the following

```
metadata = {"http://purl.org/dc/elements/1.1/": "dublin_core.xml"}
identifiers = {"DOI": "doi:10.1038/nphys1170", "ISBN": "978-3-16-148410-0"}
package_path = simple_asset_package(preservation_file="my-image.tiff", access_file="my-
↪image.jpg"
                                parent_folder=folder, Asset_Metadata=metadata, ↪
↪Identifiers=identifiers)
```

More complex assets can also be defined which contain multiple Content Objects, for example a book with multiple pages etc.

The `complex_asset_package` function takes a collection of preservation files and an optional collection of access files. It creates a single asset package with multiple content objects per Representation.

Use a `list` collection to preserve the ordering of the content objects within the asset. For example the first page of a book should be the first item added to the list.

```
preservation_files = list()
preservation_files.append("page-1.tiff")
preservation_files.append("page-2.tiff")
preservation_files.append("page-3.tiff")

access_files = list()
access_files.append("book.pdf")
```

(continues on next page)

(continued from previous page)

```
package_path = complex_asset_package(preservation_files_list=preservation_files, access_
↳ files_list=access_files,
                                parent_folder=folder)
```

15.4 Creating Packages with Multiple Representations

If you have a single preservation and access representation then `complex_asset_package` will create the package you need. If you have more than one representation of each type than you need to use `generic_asset_package`

`generic_asset_package` can be used to create as many representations as required.

`generic_asset_package` works the same way as `complex_asset_package` but instead of a list of content objects you pass a dictionary, the key is the representation name and the value is the list of files.

```
preservation_representations = dict()
preservation_representations["Master"] = ["page-1.tiff", "page-2.tiff", "page-3.tiff"]
preservation_representations["BW Master"] = ["page-1.jp2", "page-2.jp2", "page-3.jp2"]
preservation_representations["Greyscale Master"] = ["page-1.tiff", "page-2.tiff", "page-
↳ 3.tiff"]

access_representations = dict()
access_representations["Multi-Page Access"] = ["page-1.jpg", "page-2.jpg", "page-3.jpg"]
access_representations["Single Page Access"] = ["book.pdf"]

package_path = generic_asset_package(preservation_files_dict=preservation_
↳ representations, access_files_dict=access_representations, parent_folder=folder)
```

The additional keyword arguments used by `complex_asset_package` such as Title, Description etc are still available.

Preservica will render the first access representation, so the viewer you want to use needs to be the first entry in the dict. For example above if you want to use the multi-page book viewer as the default renderer, make “Multi-Page Access” the first entry, if you want the PDF viewer to be the default renderer, then make “Single Page Access” the first dict entry.

15.5 Custom Fixity Generation

By default the `simple_asset_package` and `complex_asset_package` routines will create packages which contain [SHA1](#) fixity values.

You can override this default behaviour through the use of the callback options. The pyPreservica library provides default callbacks for SHA-1, SHA256 & SHA512

- `Sha1FixityCallback`
- `Sha256FixityCallback`
- `Sha512FixityCallback`

To use one of the default callbacks

```
package_path = complex_asset_package(preservation_files_list=preservation_files, access_
↳ files_list=access_files,
                                parent_folder=folder, Preservation_files_fixity_
↳ callback=Sha512FixityCallback())
```


If you want to re-use existing externally generated fixity values for performance or integrity reasons then you can create a custom callback. The callback takes the filename and the path of the file which should have its fixity measured and should return a tuple containing the algorithm name and fixity value

```
class MyFixityCallback:
    def __call__(self, filename, full_path):
        ...
        ...
        return "SHA1", value
```

For example if your fixity values are stored in a spreadsheet (csv) files you may want something similar to:

```
class CSVFixityCallback:

    def __init__(self, csv_file):
        self.csv_file = csv_file

    def __call__(self, filename, full_path):
        with open(self.csv_file, mode='r', encoding='utf-8-sig') as csv_file:
            csv_reader = csv.DictReader(csv_file, delimiter=',')
            for row in csv_reader:
                if row['filename'] == filename:
                    fixity_value = row['file_checksum_sha256']
                    return "SHA256", fixity_value.lower()
            sha = FileHash(hashlib.sha256)
            return "SHA256", sha(full_path)
```

15.6 Bulk Package Creation

The `simple_asset_package` and `complex_asset_package` functions create a submission package containing a single Asset. If you have many single file assets to ingest you can call these functions for each file.

For example, the code fragment below will create a single Asset package for every jpg file in a directory and upload each package to Preservica.

```
path = "C:\\Jpeg-Images\\"

images = [f for f in listdir(path) if isfile(join(path, f)) and f.endswith(".jpg")]
files = [os.path.join(path, o) for o in images]

for file in files:
    package_path = simple_asset_package(preservation_file=file, parent_folder=folder)
    client.upload_zip_package(path_to_zip_package=package_path)
```

This works fine, but this will create a package for each file and an ingest workflow for each file. A more efficient way is to create a single package which contains multiple assets.

To create a multiple asset package use `multi_asset_package`, this takes a list of files and creates a package containing multiple assets which will be ingested into the same folder.

The equivalent to the code above would be:

```
path = "C:\\Jpeg-Images\\"

images = [f for f in listdir(path) if isfile(join(path, f)) and f.endswith(".jpg")]
files = [os.path.join(path, o) for o in images]

package_path = multi_asset_package(preservation_file=files, parent_folder=folder)
client.upload_zip_package(path_to_zip_package=package_path)
```

15.7 Package Examples

The following code samples show different ways of ingesting data into Preservica for different use cases.

15.7.1 Ingest a single digital file as an asset, with a progress bar during upload, delete the package after upload has completed.

```
from pyPreservica import *

upload = UploadAPI()

folder = "54308774-4822-4593-a8ad-970ca511caa0"

image = "./data/file.jpg"

# Create a simple package with 1 Asset and Representation and 1 CO
package = simple_asset_package(preservation_file=image, parent_folder=folder)

# Send the package via the S3 ingest bucket
# use the bucket name attached to the ingest workflow you want to use

bucket = "com.preservica.upload"

callback=UploadProgressCallback(package)

upload.upload_zip_package_to_S3(path_to_zip_package=package, bucket_name=bucket,
    ↳callback=callback, delete_after_upload=True)
```

15.7.2 Ingest a single digital file as an asset, with a custom asset Title and Description

```
from pyPreservica import *

upload = UploadAPI()

folder = "54308774-4822-4593-a8ad-970ca511caa0"

image = "./data/file.jpg"
```

(continues on next page)

(continued from previous page)

```

title = "The Asset Title"
description = "The Asset Description"

# Create a simple package with 1 Asset and Representation and 1 CO
package = simple_asset_package(preservation_file=image, parent_folder=folder,
    ↪Title=title, Description=description)

# Send the package via the S3 ingest bucket
# use the bucket name attached to the ingest workflow you want to use
bucket = "com.preservica.upload"
callback=UploadProgressCallback(package)
upload.upload_zip_package_to_S3(path_to_zip_package=package, bucket_name=bucket,
    ↪callback=callback, delete_after_upload=True)

```

15.7.3 Ingest each jpeg file in a directory as an individual asset

```

import glob
from pyPreservica import *

upload = UploadAPI()

folder = "54308774-4822-4593-a8ad-970ca511caa0"

directory = "./data/*.jpg"

# Create simple packages with 1 Asset and 1 CO for every file in the folder
bucket = "com.preservica.upload"
for image in glob.glob(directory):
    package = simple_asset_package(preservation_file=image, parent_folder=folder)
    upload.upload_zip_package_to_S3(path_to_zip_package=package, bucket_name=bucket)

```

15.7.4 Ingest a single digital file as an asset with a 3rd party identifier and custom metadata

```

from pyPreservica import *

upload = UploadAPI()

folder = "54308774-4822-4593-a8ad-970ca511caa0"

image = "./data/file.jpg"

# Set the Asset Title and Description

title = "My Assst Title"
description = "My Assst Description"

# Add 3rd Party Identifiers

```

(continues on next page)

(continued from previous page)

```

identifiers = {"ISBN": "123-4567-938"}

# Add Description metadata

metadata = {"https://www.example.com/metadata": "./metadata/dc.xml"}

package = simple_asset_package(preservation_file=image, parent_folder=folder,
                               Title=title, Description=description,
                               ↪ Identifiers=identifiers, Asset_Metadata=metadata)

bucket = "com.preservica.upload"

upload.upload_zip_package_to_S3(path_to_zip_package=package, bucket_name=bucket, delete_
                               ↪ after_upload=True)

```

15.7.5 Create a single Asset with 2 Representations (Preservation and Access) each Representation has 1 Content Object

```

from pyPreservica import *

upload = UploadAPI()

folder = "54308774-4822-4593-a8ad-970ca511caa0"

access_image = "./jpeg/file.jpg"
preservation_image = "./tiff/file.tif"

package = simple_asset_package(preservation_file=preservation_image, access_file=access_
                               ↪ image,
                               parent_folder=folder)

bucket = "com.preservica.upload"
upload.upload_zip_package_to_S3(path_to_zip_package=package, bucket_name=bucket, delete_
                               ↪ after_upload=True)

```

15.7.6 Create a package with 1 Asset 2 Representations (Preservation and Access) and multiple Content Objects (one for every image)

```

import glob
from pyPreservica import *

upload = UploadAPI()

folder = "54308774-4822-4593-a8ad-970ca511caa0"

access_images = "./data/*.jpg"
preservation_images = "./data2/*.tif"

```

(continues on next page)

(continued from previous page)

```

package = complex_asset_package(preservation_files_list=glob.glob(preservation_images),
                                access_files_list=glob.glob(access_images),
                                parent_folder=folder)

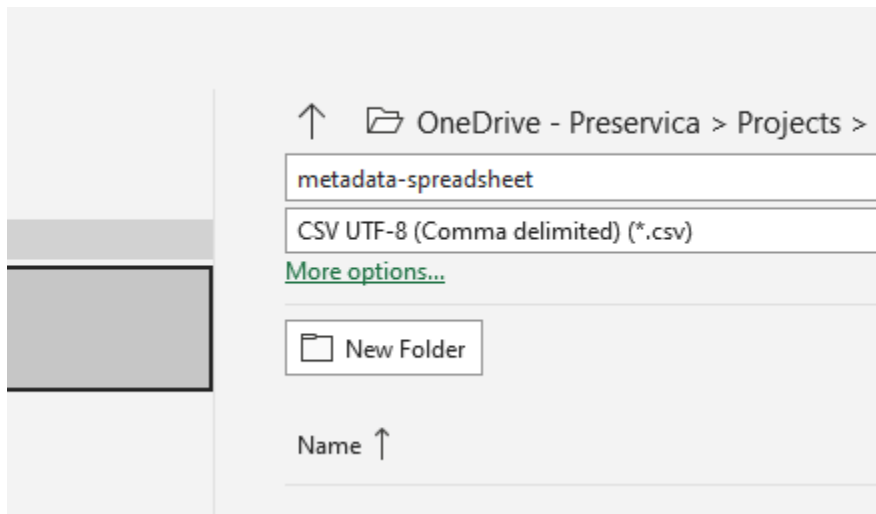
bucket = "com.preservica.upload"
upload.upload_zip_package_to_S3(path_to_zip_package=package, bucket_name=bucket, delete_
↪after_upload=True)

```

15.8 Spreadsheet Metadata

pyPreservica now provides some experimental support for working with metadata in spreadsheets. The library provides support for generating descriptive metadata XML documents for each row in a spreadsheet, creating an XSD schema for the XML documents and creating a custom transform for viewing the metadata in the UA portal along side a custom search index.

Before working with the spreadsheet it should be saved as a UTF-8 CSV document within Excel.



CSV to XML works by extracting each row of a spreadsheet and creating a single XML document for each row. The spreadsheet columns are the XML attributes.

The XML namespace and root element need to be provided. You also need to specify which column should be used to name the XML files.

```

cvs_to_xml(csv_file="my-spreadsheet.csv", root_element="Metadata", file_name_column=
↪"filename", xml_namespace="https://test.com/Metadata")

```

This will read the my-spreadsheet.csv csv file and create a set of XML documents, one for each row in the csv file. The XML files will be named after the value in the filename column.

The resulting XML documents will look like

```

<?xml version='1.0' encoding='utf-8'?>
<Metadata xmlns="https://test.com/Metadata">
  <Column1>....</Column1>
  <Column2>....</Column2>

```

(continues on next page)

(continued from previous page)

```
<Column3>....</Column3>
<Column4>....</Column4>
</Metadata>
```

You can create a XSD schema for the documents by calling

```
cvs_to_xsd(csv_file="my-spreadsheet.csv", root_element="Metadata", xml_namespace="https://
↳ /test.com/Metadata")
```

Which will generate a document `Metadata.xsd`

```
<?xml version='1.0' encoding='utf-8'?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" attributeFormDefault="unqualified
↳ " elementFormDefault="qualified"
    targetNamespace="https://test.com/Metadata">
  <xs:element name="Metadata">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:string" name="Column1" />
        <xs:element type="xs:string" name="Column2" />
        <xs:element type="xs:string" name="Column3" />
        <xs:element type="xs:string" name="Column4" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

To display the resulting metadata in the UA portal you will need a CMIS transform to tell Preservica which attributes to display. You can generate one by calling

```
cvs_to_cmis_xslt(csv_file="my-spreadsheet.csv", root_element="Metadata", title="My_
↳ Metadata Title",
    xml_namespace="https://test.com/Metadata")
```

You can also auto-generate a custom search index document which will add indexes for each column in the spreadsheet

```
csv_to_search_xml(csv_file="my-spreadsheet.csv", root_element="Metadata",
    xml_namespace="https://test.com/Metadata")
```

15.9 Ingest Web Video

pyPreservica now contains the ability to ingest web video directly from video hosting sites such as YouTube and others. To use this functionality you need to install the additional Python Project `youtube_dl`

```
$ pip install --upgrade youtube_dl
```

You can ingest video's directly with only the video site URL. You also need to tell Preservica which folder the new video asset will be ingested into.

```
upload = UploadAPI()
client = EntityAPI()
```

(continues on next page)

(continued from previous page)

```

folder = client.folder("edf403d0-04af-46b0-ab21-e7a620bfdedf")

upload.ingest_web_video(url="https://www.youtube.com/watch?v=4GCr9gljY7s", parent_
↳ folder=folder):

```

The new asset will get the title and description from youtube metadata. The asset will be given the default security tag of “open”.

The video is downloaded from the web hosting platform to the local client running the Python script and then uploaded to Preservica.

It will work with most sites that host video, for example using c-span.

```

upload = UploadAPI()
client = EntityAPI()

cspan_url = "https://www.c-span.org/video/?508691-1/ceremonial-swearing-democratic-
↳ senator-padilla"
folder = client.folder("edf403d0-04af-46b0-ab21-e7a620bfdedf")

upload.ingest_web_video(url=cspan_url, parent_folder=folder):

```

or UK parliament

```

upload = UploadAPI()
client = EntityAPI()

uk_url = "https://parliamentlive.tv/event/index/b886f44b-0e65-47bc-b506-d0e805c01f4b"
folder = client.folder("edf403d0-04af-46b0-ab21-e7a620bfdedf")

upload.ingest_web_video(url=uk_url, parent_folder=folder):

```

The asset will automatically have a title and description pulled from the original site.

You can override the default title, description and security tag with optional arguments and add 3rd party identifiers.

```

upload = UploadAPI()
client = EntityAPI()

identifier_map = {"Type": "youtube.com"}

url = "https://www.youtube.com/watch?v=4GCr9gljY7s"
title = "Preservica Cloud Edition: Keeping your digital assets safe and accessible"

folder = client.folder("edf403d0-04af-46b0-ab21-e7a620bfdedf")

upload.ingest_web_video(url=url, parent_folder=folder, Identifiers=identifier_dict,
↳ Title=title, SecurityTag="public")

```

15.10 Ingest Twitter Feeds

To use this functionality you need to install the additional Python Project tweepy

```
$ pip install --upgrade tweepy
```

The Twitter API is authenticated, this means that unlike youtube you need a set of API credentials to read tweets even if the tweets are public and you have a twitter account.

You can apply for API Consumer Keys (The basic ready only set is required) at:

<https://developer.twitter.com/>

You will need the consumer key and secret. Your twitter API keys and tokens should be guarded very carefully.

Note: Twitter no longer provides free API read access. See: <https://developer.twitter.com/>

You can harvest and ingest tweets using a single call on the upload class using `ingest_twitter_feed` method.

You should pass the name of the twitter feed you want to crawl and the number of tweets as the first two arguments. You also need to tell the call which folder you want the tweet assets to be ingested into.

The twitter API Consumer Keys can either be passed as arguments to the call or be specified in the credential.properties file or an environment variable using the keys: `TWITTER_CONSUMER_KEY` and `TWITTER_SECRET_KEY`

```
upload = UploadAPI()

twitter_name = "Preservica"
number_tweets = 25
folder_id = "77802d22-ee48-4e46-9b29-46118246cad1"
folder = entity.folder(folder_id)

upload.ingest_twitter_feed(twitter_user=twitter_name, num_tweets=number_tweets,
    ↪ folder=folder, twitter_consumer_key="xxxx", twitter_secret_key="zzzz")
```

15.11 Crawl and ingest from a filesystem

The UploadAPI class provides a mechanism for users to crawl and ingest all digital files on a filesystem. The crawl will replicate the on disk folder structure in Preservica.

You provide the function the path to the data to be ingested, an bucket connected to an ingest workflow and the Preservica collection to ingest into.

```
upload = UploadAPI()

upload.crawl_filesystem(filesystem_path="/my/path/data", bucket_name="com.bucket",
    preservica_parent="daa88307-4a0b-4962-a5a9-6a1387f9f876")
```


WORKFLOW API

The workflow API allows clients to interact with the workflow engine, you can start workflows programmatically and monitor the workflow queue etc.

Note: The Workflow API is available for Enterprise Preservica users only

Begin by importing the pyPreservica module

```
from pyPreservica import *
```

Now, create the WorkflowAPI client

```
client = WorkflowAPI()
```

16.1 Fetching Workflow Contexts

The WorkflowAPI allows users to fetch a list of workflow contexts. A workflow context is a workflow definition which has been configured and is ready to run. Workflow contexts will appear in the “Manage” tab in the admin interface under the workflow type.

To fetch list of all workflow contexts by the workflow definition identifier

```
for workflow_context in client.get_workflow_contexts("com.preservica.core.workflow.ingest"):
    print(workflow_context.workflow_name)
```

To fetch a list of all workflow contexts by type:

The list of available types are:

- Ingest
- Access
- Transformation
- DataManagement

```
for workflow_context in client.get_workflow_contexts_by_type("Ingest"):
    print(workflow_context.workflow_name)
```

16.2 Fetching Workflow Instances

A workflow instance is a workflow context which has been started and has either completed or is in progress.

Return a workflow instance by its identifier

```
workflow_instance = client.workflow_instance(instance_id)
print(workflow_instance.workflow_context_name)
print(workflow_instance.display_state)
```

Return a list of all Workflow instances, you can filter on workflow state and workflow type

Workflow States

- Aborted
- Active
- Completed
- Finished_Mixed_Outcome
- Pending
- Suspended
- Unknown
- Failed

Workflow Types

- Ingest
- Access
- Transformation
- DataManagement

```
for workflow_instance in client.workflow_instances("Completed", "Ingest"):
    print(workflow_instance)
```

16.3 Starting Workflows

Once you have a workflow context setup, you can start workflows via the API.

To start the workflow pass a workflow context object as the argument

```
client.start_workflow_instance(workflow_context)
```

If a workflow requires additional arguments or you would like to override the defaults, you can pass additional named arguments as additional parameters.

For example, to automatically start a new web crawl workflow, overriding some of the default parameters you would use:

```
workflow_context = client.get_workflow_contexts("com.preservica.core.workflow.web.crawl.
↳and.ingest")[0]

client.start_workflow_instance(workflow_context, seedUrl="preservica.com", maxDepth="8",
↳maxHops="10")
```


ADMIN API

pyPreservica 1.2 onwards now provides interfaces to the Administration and Management API

<https://eu.preservica.com/api/admin/documentation.html>

Note: Administration and Management API is a system management API for repository managers who have at least the role `ROLE_SDB_MANAGER_USER`

The Administration and Management API client is created using

```
from pyPreservica import *  
  
client = AdminAPI()
```

17.1 Metadata Management (XSD Schema's, XML Documents & XSLT Transforms)

Preservica holds XML metadata schema's, XML templates and XSLT transforms, you can access the document stores programmatically via the admin API.

To list all the XML templates use

```
from pyPreservica import *  
  
client = AdminAPI()  
  
client.xml_documents()
```

This will return a list of dictionary objects containing the template attributes, e.g.

```
from pyPreservica import *  
  
client = AdminAPI()  
  
for doc in client.xml_documents():  
    print(doc['Name'])
```

You can access the XSD schema and XSLT templates in the same way

```
from pyPreservica import *

client = AdminAPI()

for schema in client.xml_schemas():
    print(schema['Name'])
```

```
from pyPreservica import *

client = AdminAPI()

for transform in client.xml_transforms():
    print(transform['Name'])
```

Individual xml documents can be requested via their namespace URI.

For example, to save a MODS xml template held in Preservica with a given URI to a local file, use:

```
from pyPreservica import *

client = AdminAPI()

with open("mods-template.xml", encoding="utf-8", mode="wt") as f:
    f.write(client.xml_document("http://www.loc.gov/mods/v3"))
```

This now allows you to fetch a template from Preservica, update it and add it to a submission.

```
admin = AdminAPI()

dublin_core_template = admin.xml_document("http://www.openarchives.org/OAI/2.0/oai_dc/")
entity_response = xml.etree.ElementTree.fromstring(dublin_core_template)
entity_response.find(".//{http://purl.org/dc/elements/1.1/}title").text = "My Asset Title"
↪
dublin_core_metadata = xml.etree.ElementTree.tostring(entity_response).decode("utf-8")

package = simple_asset_package(preservation_file="my-image.tiff",
                               Asset_Metadata={"http://www.openarchives.org/OAI/2.0/oai_
↪dc/", dublin_core_metadata})
```

You can use similar code to fetch the XSD schema documents

```
from pyPreservica import *

client = AdminAPI()

with open("dublin-core.xsd", encoding="utf-8", mode="wt") as f:
    f.write(client.xml_schema("http://purl.org/dc/elements/1.1/"))
```

To fetch a transform you need to provide both an input URI and output URI

```
from pyPreservica import *

client = AdminAPI()
```

(continues on next page)

(continued from previous page)

```
with open("ead-cmis.xslt", encoding="utf-8", mode="wt") as f:
    f.write(client.xml_transform("urn:isbn:1-931666-22-9", "http://www.w3.org/1999/xhtml
↵"))
```

To add a new XML descriptive metadata template you can either pass an XML document held as a string or a file like object. If using a file, then make sure the file descriptor is opened in binary mode.

```
from pyPreservica import *

client = AdminAPI()

with open("my-template.xml", mode="rb") as f:
    f.write(client.add_xml_document("my-template-name", f))
```

or via a string

```
from pyPreservica import *

client = AdminAPI()

client.add_xml_document("my-template-name", xml_document)
```

To delete an existing XML template use the URI identifier

```
from pyPreservica import *

client = AdminAPI()

client.delete_xml_document("http://purl.org/dc/elements/1.1/")
```

XSD Schema's and XSLT Transforms can be added and deleted in a similar way

Using a file like object

```
from pyPreservica import *

client = AdminAPI()

with open("my-schema.xsd", mode="rb") as f:
    f.write(client.add_xml_schema(name="my-schema", description="", originalName="my-
↵schema.xsd", f))
```

or via a string

```
from pyPreservica import *

client = AdminAPI()

client.add_xml_schema(name="my-schema", description="", originalName="my-schema.xsd",
↵xml_document)
```

and deletion is via the URI

```
from pyPreservica import *

client = AdminAPI()

client.delete_xml_schema("http://purl.org/dc/elements/1.1/")
```

17.2 User Management

List all the users within the tenancy by their username

```
from pyPreservica import *

client = AdminAPI()

for username in client.all_users():
    print(username)
```

Fetch the full set of user details, such as full name, email address and roles

```
from pyPreservica import *

client = AdminAPI()

user = client.user_details(username):
print(user['FullName'])
print(user['Email'])
```

Create a CSV/Spreadsheet report containing details of all users within the tenancy, the report has the following columns, UserName, FullName, Email, Tenant, Enabled, Roles

```
from pyPreservica import *

client = AdminAPI()

client.user_report(report_name="users.csv")
```

Create new user accounts

```
from pyPreservica import *

client = AdminAPI()

username = "admin@example.com"
roles = ['SDB_MANAGER_USER', 'SDB_INGEST_USER']

user = client.add_user(username, full_name, roles)
```

Delete a user from the system

```
from pyPreservica import *
```

(continues on next page)

(continued from previous page)

```
client = AdminAPI()

client.delete_user(username)
```

Change the display name of a user

```
from pyPreservica import *

client = AdminAPI()

client.change_user_display_name(username, "New Display Name")
```

17.3 Security Tags

To get a list of all security tags in the system use:

```
from pyPreservica import *

client = AdminAPI()

tags = client.security_tags()
```

Note: This call may produce a different set of tags than the `user_security_tags()` function from the content API which only returns security tags that the current user has available.

You can generate a report of security tag frequency usage using the `pygal` library for example.

```
import pygal
from pygal.style import BlueStyle
from pyPreservica import *

client = AdminAPI()
search = ContentAPI()
security_tags = client.security_tags()
results = {}
for tag in security_tags:
    filters = {"xip.security_descriptor": tag, "xip.document_type": "IO"}
    hits = search.search_index_filter_hits(query="", filter_values=filters)
    results[tag] = hits

bar_chart = pygal.HorizontalBar(show_legend=False)
bar_chart.title = "Security Tag Frequency"
bar_chart.style = BlueStyle
bar_chart.x_title = "Number of Assets"
bar_chart.x_labels = results.keys()
bar_chart.add("Security Tag", results)

bar_chart.render_to_file("chart.svg")
```

This creates a graphical report which displays the frequency of each security tag with the ability to hover over the values.

The following calls are only available against a 6.4.x Preservica system.

To add a new security tag

```
from pyPreservica import *  
  
client = AdminAPI()  
  
tags = client.add_security_tag("my new tag")
```

and to delete a tag

```
from pyPreservica import *  
  
client = AdminAPI()  
  
tags = client.delete_security_tag("my new tag")
```

RETENTION API

<https://eu.preservica.com/api/entity/documentation.html#/%2Fretention-policies>

18.1 Retention Policies

Fetch a list of all retention policies

```
retention = RetentionAPI()

for policy in retention.policies():
    print(policy)
```

Fetch a retention policy by its name

```
retention = RetentionAPI()

policy = retention.policy_by_name("Standard Policy")
```

Create a new retention policy

```
retention = RetentionAPI()

args = dict()
args['Name'] = "API Created Policy"
args['Description'] = "Policy Description"
args['SecurityTag'] = "open"
args['StartDateField'] = "xip.created"
args['Period'] = "6"
args['PeriodUnit'] = "YEAR"
args['ExpiryAction'] = "REVIEW"
args['ExpiryActionParameters'] = "{\"EmailAddress\": [\"test@emailaddress1.com\", \"test@emailaddress2.com\"]}"
args['Restriction'] = "DELETE"
args['Assignable'] = bool(True)

policy = retention.create_policy(**args)
```

Delete a Policy

```
retention = RetentionAPI()

retention.delete_policy(policy.reference)
```

18.2 Retention Assignments

Assign a policy onto an asset

```
client = EntityAPI()
retention = RetentionAPI()

asset = client.asset("c365634e-9fcc-4ea1-b47f-077f55df9d64")

policy = retention.policy_by_name("Standard Policy")

retention_assignment = retention.add_assignments(asset, policy)
```

List the retention assignments on a asset

```
client = EntityAPI()
retention = RetentionAPI()

asset = client.asset("c365634e-9fcc-4ea1-b47f-077f55df9d64")

assignments = retention.assignments(asset)
```

Remove a policy assignment from an asset

```
client = EntityAPI()
retention = RetentionAPI()

retention_assignment = retention.remove_assignments(assignment)
```

REGISTRY API

PyPreservica provides a python interface for using the Preservation Action Registry API

<https://demo.preservica.com/Registry/par/documentation.html>

For more information on PAR see: <https://parcore.org/>

This pyPreservica PAR client will work with any PAR implementation which uses HTTP Basic Auth.

19.1 Non-Authenticated Read Access

The interfaces for reading information from the PAR are non-authenticated calls. Only a server address is required. All the interfaces for reading information return JSON documents.

The JSON documents can be converted into Python Dictionaries using the standard json library.

- Format Families

```
import json

par = PreservationActionRegistry(server="par-server.com")
json_document = par.format_families()
dict_obj = json.loads(json_document)

json_document = par.format_family('ae87efa4-cd5a-5d07-b1b7-251a4fe871c8')
dict_obj = json.loads(json_document)
```

- Preservation Action Types

```
par = PreservationActionRegistry(server="par-server.com")
json_document = par.preservation_action_types()
dict_obj = json.loads(json_document)

json_document = par.preservation_action_type('ae87efa4-cd5a-5d07-b1b7-251a4fe871c8')
dict_obj = json.loads(json_document)
```

- Properties

```
par = PreservationActionRegistry(server="par-server.com")
json_document = par.properties()
dict_obj = json.loads(json_document)
```

(continues on next page)

(continued from previous page)

```
json_document = par.property('ae87efa4-cd5a-5d07-b1b7-251a4fe871c8')
dict_obj = json.loads(json_document)
```

- Representation Formats

```
par = PreservationActionRegistry(server="par-server.com")
json_document = par.representation_format()
dict_obj = json.loads(json_document)

json_document = par.representation_formats('ae87efa4-cd5a-5d07-b1b7-251a4fe871c8')
dict_obj = json.loads(json_document)
```

- File Formats

```
par = PreservationActionRegistry(server="par-server.com")
json_document = par.file_formats()
dict_obj = json.loads(json_document)

json_document = par.file_format('ae87efa4-cd5a-5d07-b1b7-251a4fe871c8')
dict_obj = json.loads(json_document)
```

- Tools

```
par = PreservationActionRegistry(server="par-server.com")
json_document = par.tools()
dict_obj = json.loads(json_document)

json_document = par.tool('ae87efa4-cd5a-5d07-b1b7-251a4fe871c8')
dict_obj = json.loads(json_document)
```

- Preservation Action

```
par = PreservationActionRegistry(server="par-server.com")
json_document = par.preservation_actions()
dict_obj = json.loads(json_document)

json_document = par.preservation_action('ae87efa4-cd5a-5d07-b1b7-251a4fe871c8')
dict_obj = json.loads(json_document)
```

- Business Rules

```
par = PreservationActionRegistry(server="par-server.com")
json_document = par.business_rules()
dict_obj = json.loads(json_document)

json_document = par.business_rule('ae87efa4-cd5a-5d07-b1b7-251a4fe871c8')
dict_obj = json.loads(json_document)
```

- Rule Sets

```
par = PreservationActionRegistry(server="par-server.com")
json_document = par.rule_sets()
dict_obj = json.loads(json_document)

json_document = par.rule_set('ae87efa4-cd5a-5d07-b1b7-251a4fe871c8')
dict_obj = json.loads(json_document)
```


MONITOR API

This is an API for monitoring certain types of long running process within Preservica, for example OPEX ingests. You can find Swagger UI for this API at <https://us.preservica.com/api/processmonitor/documentation.html>

20.1 Monitors

Returns a generator of monitors. The ID returned for each monitor can be used as an ID parameter in other endpoints. These IDs might change between releases, so you should not persist them as permanent object links. Filters are additive, e.g. if both category and status filters are applied then only processes matching both category and status will be included.

This call returns a Generator which can be used to enumerate over all the monitor objects. The result is a monitor object which is dictionary created from the returned json.

```
client = MonitorAPI()

for monitor in client.monitors():
    print(monitor)
```

Filters can be applied to limit the returned data, for example:

```
client = MonitorAPI()

for monitor in client.monitors(category=MonitorCategory.INGEST, status=MonitorStatus.
    ↳ SUCCEEDED):
    print(monitor)
```

20.2 Messages

Returns a generator of process messages for each Monitor.

```
client = MonitorAPI()

for monitor in client.monitors(category=MonitorCategory.INGEST, status=MonitorStatus.
    ↳ SUCCEEDED):
    print(monitor)
    for message in client.messages(monitor['MonitorId']):
        print(message)
```

Messages can be filtered

```
client = MonitorAPI()

for monitor in client.monitors(category=MonitorCategory.INGEST, status=MonitorStatus.
↳ SUCCEEDED):
    print(monitor)
    for message in client.messages(monitor['MonitorId'], status=MessageStatus.ERROR):
        print(message)
```

20.3 Monitor Timeseries

Get the historical record of progress for a single monitor.

```
for monitor in client.monitors(category=MonitorCategory.INGEST, status=MonitorStatus.
↳ RUNNING):
    print(monitor)
    for series in client.timeseries(monitor['MonitorId']):
        print(series)
```

WEBHOOK API

pyPreservica now contains APIs for accessing the web hook API.

Webhooks are “user-defined HTTP callbacks”. They are triggered by some Preservica event, such as ingesting objects into the repository. When that event occurs, Preservica makes an HTTP request to the URL configured for the webhook.

Unlike the traditional process of “polling” in which a client asks the repository if anything has changed, web hooks automatically send out information to subscribed systems when certain events have happened.

To receive web hook notifications the 3rd party application requires a web server which can process HTTP POST requests.

To authenticate messages from Preservica to prevent spoofing attacks, the messages are verified through the use of a shared secret key.

The Webhook API requires the user to have at least the repository manager role, `ROLE_SDB_MANAGER_USER`

21.1 Subscribing

Before a system can receive notifications from Preservica, it must subscribe to a notification trigger.

Preservica currently supports three different triggers, “MOVED”, “SECURITY_CHANGED” and “INDEXED”.

The “Indexed” notification is sent after an object has been ingested and the full text index has been extracted, at this point the thumbnail and search contents are available.

When creating a new subscription service you need to generate a shared secret key and pass it as an argument to the subscribe method. This is used to verify the web service which will receive the web hooks.

The URL must be a publicly addressable web server.

```
webhook = WebHooksAPI()

webhook.subscribe("http://my-webhook-server.com:8080/", TriggerType.INDEXED, "my_shared_
↪secret")
```

The given URL host will need to respond to a validation challenge during the subscription request. Preservica will make a POST request to the URL with a `challengeCode` query parameter. The receiver must respond with the expected challenge response or the subscription will fail. The challenge response must take the form:

```
{
    "challengeCode": "challengeCode",
    "challengeResponse": "hexHmac256Response"
}
```

where hexHmac256Response is a hex hmac256 of the challengeCode using the shared secret as the hmac key.

If the web server is unable to correctly verify the subscription then an exception is thrown.

21.2 Listing Subscriptions

You can query the system for a list of current subscriptions for a tenancy.

```
webhook = WebHooksAPI()

json_doc = webhook.subscriptions()

print(json_doc)
```

21.3 Unsubscribe

To unsubscribe to a web hook, you need the subscription id

```
webhook = WebHooksAPI()

webhook.unsubscribe("c306c99ca3a736124fa711bec53c737d")
```

To unsubscribe to all web hooks use

```
webhook = WebHooksAPI()

webhook.unsubscribe_all()
```

21.4 Reference Web Server

To receive web hook notifications pyPreservica has provided a reference web server implementation which provides support for negotiation of the challenge request handshake during the subscription request and verification of each webhook event request.

To implement the web server, extend the base class *WebHookHandler* and implement a single method *do_WORK()* this method is called everytime Preservica calls the web hook. This method is therefore where any processing takes place. For example updating a catalogue system etc.

```
class MyWebHook(WebHookHandler):
    def do_WORK(self, json_payload):
        """
        Process the event
        """
```

The handler can then be used to create a web server, the web server should be run from the same directory as a *credential.properties* file containing the shared secret which was used to create the web hook subscription.

```
[credentials]
secret.key=my_shared_secret
```

For example a simple web hook server which prints the events to the console as they arrive would be:

```
from http.server import HTTPServer
from sys import argv
from pyPreservica import *

class MyWebHook(WebHookHandler):
    def do_WORK(self, json_payload):
        print(json_payload)

if __name__ == '__main__':

    config = configparser.ConfigParser(interpolation=configparser.
↪Interpolation())
    config.read('credentials.properties', encoding='utf-8')
    secret_key = config['credentials']['secret.key']

    if len(argv) > 1:
        arg = argv[1].split(':')
        BIND_HOST = arg[0]
        PORT = int(arg[1])

    print(f'Listening on http://{BIND_HOST}:{PORT}\n')

    httpd = HTTPServer((BIND_HOST, PORT), MyWebHook)
    httpd.secret_key = secret_key
    httpd.serve_forever()
```

The web server would then be started using:

```
$ python3 server.py 0.0.0.0:8000
```

A more interesting web hook handler might be one which downloads the thumbnail image from each Asset as it is ingested using the pyPreservica EntityAPI()

```
class MyWebHook(WebHookHandler):
    def do_WORK(self, json_payload):
        client = EntityAPI()
        for reference in list(json_payload['events']):
            ref = reference['entityRef']
            asset = client.asset(ref)
            client.thumbnail(asset, f"{ref}.jpg")
```


AUTHORITY RECORDS API

This API is used for managing the controlled vocabulary (Authority) records within Preservica.

Controlled vocabularies within Preservica are tables of records which can be linked to specific metadata attributes. Each table can consist of multiple records and each record has multiple fields.

The screenshot shows the 'Countries' table interface. At the top, it says 'Countries' and '5 fields, 245 records'. Below this is a toolbar with buttons: 'Configure Table Fields', 'Search the table...', 'Add', 'Choose File' (with 'No file chosen' text), 'Import', and 'Export'. The main table has columns: ID, Code, latitude, longitude, and Name. It lists countries from Andorra to Netherlands Antilles. To the right of each row are buttons: 'View', 'Linked Entities', a pencil icon, and an 'x' icon.

ID	Code	latitude	longitude	Name
2	AD	42.546245	1.601554	Andorra
3	AE	23.424076	53.847818	United Arab Emirates
4	AF	33.93911	67.709953	Afghanistan
5	AG	17.060816	-61.796428	Antigua and Barbuda
6	AI	18.220554	-63.068615	Anguilla
7	AL	41.153332	20.168331	Albania
8	AM	40.069099	45.038189	Armenia
9	AN	12.226079	-69.060087	Netherlands Antilles

22.1 Authority Tables

Fetch a list of all Authority tables

```
authority = AuthorityAPI()

for table in authority.tables():
    print(table)
```

Get a single Authority table by its reference

```
authority = AuthorityAPI()

table = authority.table(ref):
```

22.2 Authority Records

Fetch a record by its reference

```
authority = AuthorityAPI()

record = authority.record(reference)
```

Fetch all records from a table

```
authority = AuthorityAPI()

table = authority.table(ref):

for record in authority.records(table):
    print(record)
```

Add a new authority record to an existing table, the record is a Python dictionary object

```
authority = AuthorityAPI()

table = authority.table(ref):

record = {"id": "6", "Code": "BE", "Latitude": "50.503887", "Longitude": "4.469936",
↪ "Name": "Belgium"}

authority.add_record(table, record)
```

Adding records from a CSV document

```
authority = AuthorityAPI()

table = authority.table(ref):

authority.add_records(table, "countries.csv")
```

If the CSV document was saved from a MS Excel workbook, then the encoding should be set to utf-8-sig

```
authority = AuthorityAPI()

table = authority.table(ref):

authority.add_records(table, "countries.csv", encoding="utf-8-sig")
```

Deleting Records from a table by its reference

```
authority = AuthorityAPI()

table = authority.table(ref):

authority.delete_record(table, reference)
```

To delete all records from a table


```
authority = AuthorityAPI()

table = authority.table(ref):

for record in authority.records(table):
    authority.delete_record(table, record['ref'])
```


EXAMPLE APPLICATIONS

Updating a descriptive metadata element value

If you need to bulk update metadata values the following script will check every asset in a folder given by the “folder-uuid” and find the matching descriptive metadata document by its namespace “your-xml-namespace”. It will then find a particular element in the xml document “your-element-name” and update its value.

```
from xml.etree import ElementTree
from pyPreservica import *
client = EntityAPI()
folder = client.folder("folder-uuid")
next_page = None
while True:
    children = client.children(folder.reference, maximum=10, next_page=next_page)
    for entity in children.results:
        if entity.entity_type is EntityAPI.EntityType.ASSET:
            asset = client.asset(entity.reference)
            for url, schema in asset.metadata.items():
                if schema == "your-xml-namespace":
                    xml_document = ElementTree.fromstring(client.metadata(url))
                    field_with_error = xml_document.find('.//{your-xml-namespace}your-
↪element-name')
                    if hasattr(field_with_error, 'text'):
                        if field_with_error.text == "Old Value":
                            field_with_error.text = "New Value"
                            asset = client.update_metadata(asset, schema, ElementTree.
↪tostring(xml_document, encoding='UTF-8', xml_declaration=True).decode("utf-8"))
                            print("Updated asset: " + asset.title)
            if not children.has_more:
                break
            else:
                next_page = children.next_page
```

The following script does the same thing as above but uses the function descendants() rather than children(). The difference is that descendants() does the paging of results internally and combined with a filter() on the lazy iterator provides a version which does not need the additional while loop or if statement!

```
client = EntityAPI()
folder = client.folder("folder-uuid")
for child_asset in filter(only_assets, client.descendants(folder.reference)):
    asset = client.asset(child_asset.reference)
    document = ElementTree.fromstring(client.metadata_for_entity(asset, "your-xml-
```

(continues on next page)

(continued from previous page)

```

↪namespace"))
    field_with_error = document.find('.//{your-xml-namespace}your-element-name')
    if hasattr(field_with_error, 'text'):
        if field_with_error.text == "Old Value":
            field_with_error.text = "New Value"
            new_xml = ElementTree.tostring(document, encoding='UTF-8', xml_
↪declaration=True).decode("utf-8")
            asset = client.update_metadata(asset, "your-xml-namespace", new_xml)
            print("Updated asset: " + asset.title)

```

Adding Metadata from a Spreadsheet

One common use case which can be solved with pyPreservica is adding descriptive metadata to existing Preservica assets or folders using metadata held in a spreadsheet. Normally each column in the spreadsheet contains a metadata attribute and each row represents a different asset.

The following is a short python script which uses pyPreservica to update assets within Preservica with Dublin Core Metadata held in a spreadsheet.

The spreadsheet should contain a header row. The column name in the header row should start with the text “dc:” to be included. There should be one column called “assetId” which contains the reference id for the asset to be updated.

The metadata should be saved as a UTF-8 CSV file called dublincore.csv

```

import xml
import csv
from pyPreservica import *

OAI_DC = "http://www.openarchives.org/OAI/2.0/oai_dc/"
DC = "http://purl.org/dc/elements/1.1/"
XSI = "http://www.w3.org/2001/XMLSchema-instance"

entity = EntityAPI()

headers = list()
with open('dublincore.csv', encoding='utf-8-sig', newline='') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        for header in row:
            headers.append(header)
        break
    if 'assetId' in headers:
        for row in reader:
            assetID = None
            xml_object = xml.etree.ElementTree.Element('oai_dc:dc', {"xmlns:oai_dc": OAI_
↪DC, "xmlns:dc": DC, "xmlns:xsi": XSI})
            for value, header in zip(row, headers):
                if header.startswith('dc:'):
                    xml.etree.ElementTree.SubElement(xml_object, header).text = value
                elif header.startswith('assetId'):
                    assetID = value
            xml_request = xml.etree.ElementTree.tostring(xml_object, encoding='utf-8',
↪xml_declaration=True).decode('utf-8')
            asset = entity.asset(assetID)

```

(continues on next page)

(continued from previous page)

```

        entity.add_metadata(asset, OAI_DC, xml_request)
    else:
        print("The CSV file should contain a assetId column containing the Preservica_
        ↳ identifier for the asset to be updated")

```

Creating Searchable Transcripts from Oral Histories

The following is an example python script which uses a 3rd party Machine Learning API to automatically generate a text transcript from an audio file such as a WAVE file. The transcript is then uploaded to Preservica, is stored as metadata attached to an asset and indexed so that the audio or oral history is searchable.

This example uses the AWS <https://aws.amazon.com/transcribe/> service, but other AI APIs are also available. AWS provides a free tier <https://aws.amazon.com/free/> to allow you to try the service for no cost.

This python script does require a set of AWS credentials to use the AWS transcribe service.

The python script downloads a WAV file using its reference, uploads it to AWS S3 and then starts the transcription service, when the transcript is available it creates a metadata document containing the text and uploads it to Preservica.:

```

import os, time, uuid, xml, boto3, requests
from pyPreservica import *

BUCKET = "com.my.transcribe.bucket"
AWS_KEY = '.....'
AWS_SECRET = '.....'
REGION = 'eu-west-1'
## download the file to the local machine
client = EntityAPI()
asset = client.asset('91c73c95-a298-448c-a5a3-2295e5052be3')
client.download(asset, f"{asset.reference}.wav")
# upload the file to AWS
s3_client = boto3.client('s3', region_name=REGION, aws_access_key_id=AWS_KEY, aws_secret_
↳ access_key=AWS_SECRET)
response = s3_client.upload_file(f"{asset.reference}.wav", BUCKET, f"{asset.reference}")
# Start the transcription service
transcribe = boto3.client('transcribe', region_name=REGION, aws_access_key_id=KEY, aws_
↳ secret_access_key=SECRET)
job_name = str(uuid.uuid4())
job_uri = f"https://s3-{REGION}.amazonaws.com/{BUCKET}/{asset.reference}"
transcribe.start_transcription_job(TranscriptionJobName=job_name, Media={'MediaFileUri
↳ ': job_uri}, MediaFormat='wav', LanguageCode='en-US')
while True:
    status = transcribe.get_transcription_job(TranscriptionJobName=job_name)
    if status['TranscriptionJob']['TranscriptionJobStatus'] in ['COMPLETED', 'FAILED']:
        break
    print("Still working on the transcription...")
    time.sleep(5)
# upload the transcript text to Preservica
if status['TranscriptionJob']['TranscriptionJobStatus'] == 'COMPLETED':
    result_url = status['TranscriptionJob']['Transcript']['TranscriptFileUri']
    json = requests.get(result_url).json()
    text = json['results']['transcripts'][0]['transcript']
    xml_object = xml.etree.ElementTree.Element('tns:Transcript', {"xmlns:tns": "https://
↳ aws.amazon.com/transcribe/"})

```

(continues on next page)

(continued from previous page)

```
xml.etree.ElementTree.SubElement(xml_object, "Transcription").text = text
xml_request = xml.etree.ElementTree.tostring(xml_object, encoding='utf-8', xml_
↪declaration=True).decode('utf-8')
client.add_metadata(asset, "https://aws.amazon.com/transcribe/", xml_request) #
↪add the xml transcript
s3_client.delete_object(Bucket=BUCKET, Key=asset.reference) # delete the temp file
↪from s3
os.remove(f"{asset.reference}.wav") # delete the local copy
```

DEVELOPER INTERFACE

24.1 Entity API

This part of the documentation covers all the interfaces of pyPreservica *EntityAPI* object.

```
class pyPreservica.EntityAPI(username: str | None = None, password: str | None = None, tenant: str | None = None, server: str | None = None, use_shared_secret: bool = False, two_fa_secret_key: str | None = None, protocol: str = 'https')
```

A class for the Preservica Repository web services Entity API

<https://us.preservica.com/api/entity/documentation.html>

The EntityAPI allows users to interact with the Preservica repository

asset(reference)

Returns an asset object back by its internal reference identifier

Parameters

reference (str) – The unique identifier for the asset usually its uuid

Returns

The Asset object

Return type

Asset

Raises

RuntimeError – if the identifier is incorrect

folder(reference)

Returns a folder object back by its internal reference identifier

Parameters

reference (str) – The unique identifier for the asset usually its uuid

Returns

The Folder object

Return type

Folder

Raises

RuntimeError – if the identifier is incorrect

content_object(reference)

Returns a content object back by its internal reference identifier

Parameters

reference (*str*) – The unique identifier for the asset usually its uuid

Returns

The content object

Return type

ContentObject

Raises

RuntimeError – if the identifier is incorrect

entity(*entity_type*, *reference*)

Returns an generic entity based on its reference identifier

Parameters

- **entity_type** (*EntityType*) – The type of entity
- **reference** (*str*) – The unique identifier for the entity

Returns

The entity either Asset, Folder or ContentObject

Return type

Entity

Raises

RuntimeError – if the identifier is incorrect

save(*entity*)

Updates the title and description of an entity The security tag and parent are not saved via this method call

Parameters

entity (*Entity*) – The entity (asset, folder, content_object) to be updated

Returns

The updated entity

Return type

Entity

security_tag_async(*entity*, *new_tag*)

Change the security tag of an asset or folder This is a non blocking call which returns immediately.

Parameters

- **entity** (*Entity*) – The entity (asset, folder) to be updated
- **new_tag** (*str*) – The new security tag to be set on the entity

Returns

A progress id which can be used to monitor the workflow

Return type

str

security_tag_sync(*entity*, *new_tag*)

Change the security tag of an asset or folder This is a blocking call which returns after all entities have been updated.

Parameters

- **entity** (*Entity*) – The entity (asset, folder) to be updated

- **new_tag** (*str*) – The new security tag to be set on the entity

Returns

The updated entity

Return type

Entity

create_folder(*title, description, security_tag, parent=None*)

Create a new folder in the repository below the specified parent folder. If parent is missing or None then a root level folder is created.

Parameters

- **title** (*str*) – The title of the new folder
- **description** (*str*) – The description of the new folder
- **security_tag** (*str*) – The security tag of the new folder
- **parent** (*str*) – The identifier for the parent folder

Returns

The new folder object

Return type

Folder

representations(*asset*)

Return a set of representations for the asset

Representations are used to define how the information object are composed in terms of technology and structure.

Parameters

asset (*Asset*) – The asset containing the required representations

Returns

Set of Representation objects

Return type

set(*Representation*)

content_objects(*representation*)

Return a list of content objects for a representation

Parameters

representation (*Representation*) – The representation

Returns

List of content objects

Return type

list(*ContentObject*)

generations(*content_object*)

Return a list of Generation objects for a content object

Parameters

content_object (*ContentObject*) – The content object

Returns

list of generations

Return typelist(*Generation*)**bitstream_content**(*bitstream*, *filename*)

Downloads the bitstream object to a local file

Parameters

- **bitstream** (*Bitstream*) – The content object
- **filename** (*str*) – The name of the file the bytes are written to

Returns

the number of bytes written

Return type

int

identifiers_for_entity(*entity*)

Return a set of identifiers which belong to the entity

Parameters**entity** (*Entity*) – The entity**Returns**

Set of identifiers as tuples

Return type

set(Tuple)

identifier(*identifier_type*, *identifier_value*)

Return a set of entities with external identifiers which match the type and value

Parameters

- **identifier_type** (*str*) – The identifier type
- **identifier_value** (*str*) – The identifier value

Returns

Set of entity objects which have a reference and title attribute

Return typeset(*Entity*)**add_identifier**(*entity*, *identifier_type*, *identifier_value*)

Add a new external identifier to an Entity object

Parameters

- **entity** (*Entity*) – The entity the identifier is added to
- **identifier_type** (*str*) – The identifier type
- **identifier_value** (*str*) – The identifier value

Returns

An internal id for this external identifier

Return type

str

delete_identifiers(*entity*, *identifier_type=None*, *identifier_value=None*)

Delete identifiers on an Entity object

Parameters

- **entity** (*Entity*) – The entity the identifiers are deleted from
- **identifier_type** (*str*) – The identifier type
- **identifier_value** (*str*) – The identifier value

Returns

entity

Return type

Entity

metadata(*uri*)

Fetch the metadata document by its identifier, this is the key from the entity metadata map

Parameters

uri (*str*) – The metadata identifier

Returns

An XML document as a string

Return type

str

metadata_for_entity(*entity*, *schema*)

Fetch the first metadata document which matches the schema URI from an entity

Parameters

- **entity** (*Entity*) – The entity containing the metadata
- **schema** (*str*) – The metadata schema URI

Returns

The first XML document on the entity matching the schema URI

Return type

str

add_metadata(*entity*, *schema*, *data*)

Add a new descriptive XML document to an entity

Parameters

- **entity** (*Entity*) – The entity to add the metadata to
- **schema** (*str*) – The metadata schema URI
- **data** (*data*) – The XML document as a string or as a file bytes

Returns

The updated Entity

Return type

Entity

update_metadata(*entity*, *schema*, *data*)

Update an existing descriptive XML document on an entity

Parameters

- **entity** ([Entity](#)) – The entity to add the metadata to
- **schema** (*str*) – The metadata schema URI
- **data** (*data*) – The XML document as a string or as a file bytes

Returns

The updated Entity

Return type

[Entity](#)

delete_metadata(*entity, entity, schema*)

Delete an existing descriptive XML document on an entity by its schema This call will delete all fragments with the same schema

Parameters

- **entity** ([Entity](#)) – The entity to add the metadata to
- **schema** (*str*) – The metadata schema URI

Returns

The updated Entity

Return type

[Entity](#)

move_sync(*entity, dest_folder*)

Move an entity (asset or folder) to a new folder This call blocks until the move is complete

Parameters

- **entity** ([Entity](#)) – The entity to move either asset or folder
- **dest_folder** ([Entity](#)) – The new destination folder. This can be None to move a folder to the root of the repository

Returns

The updated entity

Return type

[Entity](#)

move_async(*entity, dest_folder*)

Move an entity (asset or folder) to a new folder This call returns immediately and does not block

Parameters

- **entity** ([Entity](#)) – The entity to move either asset or folder
- **dest_folder** ([Entity](#)) – The new destination folder. This can be None to move a folder to the root of the repository

Returns

Progress ID token

Return type

str

move(*entity, dest_folder*)

Move an entity (asset or folder) to a new folder This call is an alias for the move_sync (blocking) method.

Parameters

- **entity** (*Entity*) – The entity to move either asset or folder
- **dest_folder** (*Entity*) – The new destination folder. This can be None to move a folder to the root of the repository

Returns

The updated entity

Return type

Entity

children(*folder*, *maximum=50*, *next_page=None*)

Return the child entities of a folder one page at a time. The caller is responsible for requesting the next page of results.

Parameters

- **folder** (*str*) – The parent folder reference, None for the children of root folders
- **maximum** (*int*) – The maximum size of the result set in each page
- **next_page** (*str*) – A URL for the next page of results

Returns

A set of entity objects

Return type

set(*Entity*)

descendants(*folder*)

Return the immediate child entities of a folder using a lazy iterator. The paging is done internally using a default page size of 50 elements. Callers can iterate over the result to get all children with a single call.

Parameters

folder (*str*) – The parent folder reference, None for the children of root folders

Returns

A set of entity objects (Folders and Assets)

Return type

set(*Entity*)

all_descendants(*folder*)

Return all child entities recursively of a folder or repository down to the assets using a lazy iterator. The paging is done internally using a default page size of 25 elements. Callers can iterate over the result to get all children with a single call.

Parameters

folder (*str*) – The parent folder reference, None for the children of root folders

Returns

A set of entity objects (Folders and Assets)

Return type

set(*Entity*)

delete_asset(*asset*, *operator_comment*, *supervisor_comment*)

Initiate and approve the deletion of an asset.

Parameters

- **asset** (*Asset*) – The asset to delete
- **operator_comment** (*str*) – The comments from the operator which are added to the logs

- **supervisor_comment** (*str*) – The comments from the supervisor which are added to the logs

Returns

The asset reference

Return type

str

delete_folder(*asset, operator_comment, supervisor_comment*)

Initiate and approve the deletion of a folder.

Parameters

- **asset** (*Folder*) – The folder to delete
- **operator_comment** (*str*) – The comments from the operator which are added to the logs
- **supervisor_comment** (*str*) – The comments from the supervisor which are added to the logs

Returns

The folder reference

Return type

str

thumbnail(*entity, filename, size=Thumbnail.LARGE*)

Get the thumbnail image for an asset or folder

Parameters

- **entity** (*Entity*) – The entity
- **filename** (*str*) – The file the image is written to
- **size** (*Thumbnail*) – The size of the thumbnail image

Returns

The filename

Return type

str

download(*entity, filename*)

Download the first generation of the access representation of an asset

Parameters

- **entity** (*Entity*) – The entity
- **filename** (*str*) – The file the image is written to
- **size** (*Thumbnail*) – The size of the thumbnail image

Returns

The filename

Return type

str

updated_entities(*previous_days: int = 1*)

Fetch a list of entities which have changed (been updated) over the previous n days.

This method uses a generator function to make repeated calls to the server for every page of results.

Parameters

previous_days (*int*) – The number of days to check for changes.

Returns

A list of entities

Return type

list

all_events()

Returns a list of events for the user's tenancy

This method uses a generator function to make repeated calls to the server for every page of results.

Returns

A list of events

Return type

list

entity_events(*entity*: [Entity](#))

Returns a list of event actions performed against this entity

This method uses a generator function to make repeated calls to the server for every page of results.

Parameters

entity ([Entity](#)) – The entity

Returns

A list of events

Return type

list

add_thumbnail(*entity*: [Entity](#), *image_file*: *str*)

Set the thumbnail for the entity to the uploaded image

Supported image formats are png, jpeg, tiff, gif and bmp. The image must be 10MB or less in size.

Parameters

- **entity** ([Entity](#)) – The entity
- **image_file** (*str*) – The path to the image

remove_thumbnail(*entity*: [Entity](#))

Remove the thumbnail for the entity to the uploaded image

Parameters

image_file (*str*) – The path to the image

replace_generation_sync(*content_object*: [ContentObject](#), *file_name*: *str*, *fixity_algorithm*, *fixity_value*)

Replace the last active generation of a content object with a new digital file.

Starts the workflow and blocks until the workflow completes.

Parameters

- **content_object** ([ContentObject](#)) – The content object to replace
- **file_name** (*str*) – The path to the new content object
- **fixity_algorithm** (*str*) – Optional fixity algorithm
- **fixity_value** (*str*) – Optional fixity value

Returns

Completed workflow status

Return type

str

replace_generation_async(*content_object*: [ContentObject](#), *file_name*: str, *fixity_algorithm*, *fixity_value*)

Replace the last active generation of a content object with a new digital file.

Starts the workflow and returns a process ID

Parameters

- **content_object** ([ContentObject](#)) – The content object to replace
- **file_name** (str) – The path to the new content object
- **fixity_algorithm** (str) – Optional fixity algorithm
- **fixity_value** (str) – Optional fixity value

Returns

Process ID

Return type

str

get_async_progress(*pid*: str)

Return the status of a running process

Parameters

str (*pid*) – The progress ID

Returns

Workflow status

Return type

str

export_opex(*entity*: [Entity](#), ***kwargs*)

Initiates export of the entity and downloads the opex package

Parameters

- **entity** ([Entity](#)) – The entity to export Asset or Folder
- **IncludeContent** (str) – “Content”, “NoContent”
- **IncludeMetadata** (str) – “Metadata”, “NoMetadata”, “MetadataWithEvents”
- **IncludedGenerations** (str) – “LatestActive”, “AllActive”, “All”
- **IncludeParentHierarchy** (str) – “true”, “false”

Returns

The path to the opex ZIP file

Return type

str

class pyPreservica.[Generation](#)

Generations represent changes to content objects over time, as formats become obsolete new generations may need to be created to make the information accessible.

original

original generation (True or False)

active

active generation (True or False)

format_group

format for this generation

effective_date

effective date generation

bitstreams

list of Bitstream objects

properties

list of technical properties each property is dict object containing PUID, PropertyName and Value

formats

list of technical formats each format is dict object containing PUID, FormatName and FormatVersion

class pyPreservica.Bitstream

Bitstreams represent the actual computer files as ingested into Preservica, i.e. the TIFF photograph or the PDF document

filename

The filename of the original bitstream

length

The file size in bytes of the original Bitstream

fixity

Dictionary object of fixity values for this bitstream, the key is the algorithm name and the value is the fixity value

class pyPreservica.Representation

Representations are used to define how the information object are composed in terms of technology and structure.

rep_type

The type of representation

name

The name of representation

asset

The asset the representation belongs to

class pyPreservica.Entity

Entity is the base class for assets, folders and content objects They all have the following attributes

reference

The unique internal reference for the entity

title

The title of the entity

description

The description of the entity

security_tag

The security tag of the entity

parent

The unique internal reference for this entity's parent object

The parent of an Asset is always a Folder

The parent of a Folder is always a Folder or None for a folder at the root of the repository

The parent of a Content Object is always an Asset

metadata

A map of descriptive metadata attached to the entity.

The key of the map is the metadata identifier used to retrieve the metadata document and the value is the schema URI

entity_type

Assets have entity type EntityType.ASSET

Folders have entity type EntityType.FOLDER

Content Objects have entity type EntityType.CONTENT_OBJECT

class pyPreservica.Asset

Asset represents the information object or intellectual unit of information within the repository.

reference

The unique internal reference for the asset

title

The title of the asset

description

The description of the asset

security_tag

The security tag of the asset

parent

The unique internal reference for this asset's parent folder

metadata

A dict of descriptive metadata attached to the asset.

The key of the dict is the metadata identifier used to retrieve the metadata document and the value is the schema URI

entity_type

Assets have entity type `EntityType.ASSET`

class pyPreservica.Folder

Folder represents the structure of the repository and contains both Assets and Folder objects.

reference

The unique internal reference for the folder

title

The title of the folder

description

The description of the folder

security_tag

The security tag of the folder

parent

The unique internal reference for this folder's parent folder

metadata

A map of descriptive metadata attached to the folder.

The key of the map is the metadata identifier used to retrieve the metadata document and the value is the schema URI

entity_type

Assets have entity type `EntityType.FOLDER`

class pyPreservica.ContentObject

ContentObject represents the internal structure of an asset.

reference

The unique internal reference for the content object

title

The title of the content object

description

The description of the content object

security_tag

The security tag of the content object

parent

The unique internal reference for this content object parent asset

metadata

A map of descriptive metadata attached to the content object.

The key of the map is the metadata identifier used to retrieve the metadata document and the value is the schema URI

entity_type

Content objects have entity type `EntityType.CONTENT_OBJECT`

class pyPreservica.**EntityType**(*value*)

Enumeration of the Entity Types

class pyPreservica.**RelationshipDirection**(*value*)

An enumeration.

class pyPreservica.**IntegrityCheck**(*check_type, success, date, adapter, fixed, reason*)

Class to hold information about completed integrity checks

24.2 Content API

This part of the documentation covers all the interfaces of pyPreservica [UploadAPI](#) object.

class pyPreservica.**ContentAPI**

object_details(*entity_type, reference*)

Return a list of all the indexed fields in the Preservica search index.

Parameters

- **entity_type** (*str*) – Entity type, either “IO” or “SO”
- **reference** (*str*) – Entity reference

Returns

object attributes

Return type

dict

indexed_fields()

Return a list of all the indexed fields in the Preservica search index.

Returns

list of index field names

Return type

list

simple_search_list(*query: str = '%', page_size: int = 10, *args*)

Search Preservica using a simple search term across all indexed fields, the results are returned as generator

Parameters

- **query** (*str*) – Query term

- **page_size** (*int*) – Number of results fetched between server calls
- **args** (*tuple*) – index names to include in the result

Returns

list of search result hits

Return type

list

simple_search_csv(*query: str = '%', csv_file='search.csv', *args*)

Search Preservica using a simple search term across all indexed fields, output the results to a csv file.

Parameters

- **query** (*str*) – Query term
- **page_size** (*int*) – Number of results fetched between server calls
- **args** (*tuple*) – index names to include in the result

24.3 Upload API

This part of the documentation covers all the interfaces of pyPreservica [UploadAPI](#) object.

pyPreservica.simple_asset_package(*preservation_file=None, access_file=None, export_folder=None, parent_folder=None, compress=True, **kwargs*)

Create a Preservica package containing a single Asset from a single preservation file and an optional access file. The Asset contains one Content Object for each representation.

If only the preservation file is provided the asset has one representation

Parameters

- **preservation_file** (*str*) – Path to the preservation file
- **access_file** (*str*) – Path to the access file
- **export_folder** (*str*) – The package location folder
- **parent_folder** ([Folder](#)) – The folder to ingest the asset into
- **compress** (*bool*) – Compress the ZIP file
- **Title** (*str*) – Asset Title
- **Description** (*str*) – Asset Description
- **SecurityTag** (*str*) – Asset SecurityTag
- **CustomType** (*str*) – Asset CustomType
- **Preservation_Content_Title** (*str*) – Title of the Preservation Representation Content Object
- **Preservation_Content_Description** (*str*) – Description of the Preservation Representation Content Object
- **Access_Content_Title** (*str*) – Title of the Access Representation Content Object
- **Access_Content_Description** (*str*) – Description of the Access Representation Content Object
- **Asset_Metadata** (*dict*) – Dictionary of Asset metadata documents

- **Identifiers** (*dict*) – Dictionary of Asset rd party identifiers

`pyPreservica.complex_asset_package(preservation_files_list=None, access_files_list=None, export_folder=None, parent_folder=None, compress=True, **kwargs)`

Create a Preservica package containing a single Asset from a multiple preservation files and optional access files. The Asset contains multiple Content Objects within each representation.

If only the preservation files are provided the asset has one representation

param list preservation_files_list

Paths to the preservation files

param list access_files_list

Paths to the access files

param str export_folder

The package location folder

param Folder parent_folder

The folder to ingest the asset into

param bool compress

Compress the ZIP file

param str Title

Asset Title

param str Description

Asset Description

param str SecurityTag

Asset SecurityTag

param str CustomType

Asset CustomType

param str Preservation_Content_Title

Title of the Preservation Representation Content Object

param str Preservation_Content_Description

Description of the Preservation Representation Content Object

param str Access_Content_Title

Title of the Access Representation Content Object

param str Access_Content_Description

Description of the Access Representation Content Object

param dict Asset_Metadata

Dictionary of Asset metadata documents

param dict Identifiers

Dictionary of Asset rd party identifiers

optional kwargs map 'Title' Asset Title 'Description' Asset Description 'SecurityTag' Asset Security Tag 'CustomType' Asset Type 'Preservation_Content_Title' Content Object Title of the Preservation Object 'Preservation_Content_Description' Content Object Description of the Preservation Object 'Access_Content_Title' Content Object Title of the Access Object 'Access_Content_Description' Content Object Description of the Access Object 'Preservation_Generation_Label' Generation Label for the Preservation Object 'Access_Generation_Label' Generation Label for the Access Object 'Asset_Metadata' Map of metadata

schema/documents to add to asset 'Identifiers' Map of asset identifiers 'Preservation_files_fixity_callback' Callback to allow external generated fixity values 'Access_files_fixity_callback' Callback to allow external generated fixity values 'IO_Identifier_callback' Callback to allow external generated Asset identifier 'Preservation_Representation_Name' Name of the Preservation Representation 'Access_Representation_Name' Name of the Access Representation

```
pyPreservica.cvs_to_xml(csv_file, xml_namespace, root_element, file_name_column='filename',
                        export_folder=None, additional_namespaces=None)
```

Export the rows of a CSV file as XML metadata documents which can be added to Preservica assets

Parameters

- **csv_file** (*str*) – Path to the csv file
- **xml_namespace** (*str*) – The XML namespace for the created XML documents
- **root_element** (*str*) – The root element for the XML documents
- **file_name_column** (*str*) – The CSV column which should be used to name the xml files
- **export_folder** (*str*) – The path to the export folder
- **additional_namespaces** (*dict*) – A map of prefix, uris to use as additional namespaces

```
class pyPreservica.UploadAPI(username: str | None = None, password: str | None = None, tenant: str | None = None, server: str | None = None, use_shared_secret: bool = False,
                             two_fa_secret_key: str | None = None, protocol: str = 'https')
```

```
ingest_tweet(twitter_user=None, tweet_id: int = 0, twitter_consumer_key=None, twitter_secret_key=None,
              folder=None, callback=None, **kwargs)
```

Ingest tweets from a twitter stream by twitter username

Parameters

- **tweet_id**
- **twitter_user** (*str*) – Twitter Username
- **twitter_consumer_key** (*str*) – Optional asset title
- **twitter_secret_key** (*str*) – Optional asset description
- **folder** (*str*) – Folder to ingest into
- **callback** (*callback*) – Optional upload progress callback

Raises

RuntimeError –

```
ingest_twitter_feed(twitter_user=None, num_tweets: int = 25, twitter_consumer_key=None,
                    twitter_secret_key=None, folder=None, callback=None, **kwargs)
```

Ingest tweets from a twitter stream by twitter username

Parameters

- **twitter_user** (*str*) – Twitter Username
- **num_tweets** (*int*) – The number of tweets from the stream
- **twitter_consumer_key** (*str*) – Optional asset title
- **twitter_secret_key** (*str*) – Optional asset description
- **folder** (*str*) – Folder to ingest into
- **callback** (*callback*) – Optional upload progress callback

Raises**RuntimeError** –**ingest_web_video**(*url=None, parent_folder=None, **kwargs*)

Ingest a web video such as YouTube etc based on the URL

Parameters

- **url** (*str*) – URL to the youtube video
- **parent_folder** (**Folder**) – The folder to ingest the video into
- **Title** (*str*) – Optional asset title
- **Description** (*str*) – Optional asset description
- **SecurityTag** (*str*) – Optional asset security tag
- **Identifiers** (*dict*) – Optional asset 3rd party identifiers
- **Asset_Metadata** (*dict*) – Optional asset additional descriptive metadata
- **callback** (*callback*) – Optional upload progress callback

Raises**RuntimeError** –**upload_buckets**()

Get a list of available upload buckets

Returns

dict of bucket names and regions

upload_credentials(*location_id: str*)

Retrieves temporary upload credentials (Amazon STS, or Azure SAS) for this location.

Returns

dict

upload_locations()

Upload locations are configured on the Sources page as 'SIP Upload'. :return: dict

upload_zip_package(*path_to_zip_package, folder=None, callback=None, delete_after_upload=False*)

Uploads a zip file package directly to Preservica and starts an ingest workflow

Parameters

- **path_to_zip_package** (*str*) – Path to the package
- **folder** (**Folder**) – The folder to ingest the package into
- **callback** (*Callable*) – Optional callback to allow the callee to monitor the upload progress
- **delete_after_upload** (*bool*) – Delete the local copy of the package after the upload has completed

Returns

preservica-progress-token to allow the workflow progress to be monitored

Return type

str

Raises**RuntimeError** –

upload_zip_package_to_Azure(*path_to_zip_package, container_name, folder=None, delete_after_upload=False, show_progress=False*)

Uploads a zip file package to an Azure container connected to a Preservica Cloud System

Parameters

- **path_to_zip_package** (*str*) – Path to the package
- **container_name** (*str*) – container connected to the ingest workflow
- **folder** (*Folder*) – The folder to ingest the package into
- **delete_after_upload** (*bool*) – Delete the local copy of the package after the upload has completed

upload_zip_package_to_S3(*path_to_zip_package, bucket_name, folder=None, callback=None, delete_after_upload=False*)

Uploads a zip file package to an S3 bucket connected to a Preservica Cloud System

Parameters

- **path_to_zip_package** (*str*) – Path to the package
- **bucket_name** (*str*) – Bucket connected to an ingest workflow
- **folder** (*Folder*) – The folder to ingest the package into
- **callback** (*Callable*) – Optional callback to allow the callee to monitor the upload progress
- **delete_after_upload** (*bool*) – Delete the local copy of the package after the upload has completed

upload_zip_to_Source(*path_to_zip_package, container_name, folder=None, delete_after_upload=False, show_progress=False*)

Uploads a zip file package to either an Azure container or S3 bucket depending on the Preservica system deployment

Parameters

- **path_to_zip_package** (*str*) – Path to the package
- **container_name** (*str*) – container connected to the ingest workflow
- **folder** (*Folder*) – The folder to ingest the package into
- **delete_after_upload** (*bool*) – Delete the local copy of the package after the upload has completed
- **show_progress** (*bool*) – Show upload progress bar

24.4 Retention Management API

<https://eu.preservica.com/api/entity/documentation.html#/%2Fretention-policies>

class pyPreservica.RetentionPolicy(*name: str, reference: str*)

class pyPreservica.RetentionAssignment(*entity_reference: str, policy_reference: str, api_id: str, start_date, expired=False*)

```
class pyPreservica.RetentionAPI(username=None, password=None, tenant=None, server=None,
                                use_shared_secret=False, two_fa_secret_key: str | None = None, protocol:
                                str = 'https')
```

```
add_assignments(entity: Entity, policy: RetentionPolicy) → RetentionAssignment
```

Assign a retention policy to an Asset.

Parameters

- **entity** (*Entity*) – The Preservica Entity to assign a policy to
- **policy** (*RetentionPolicy*) – The RetentionAssignment

Returns

The RetentionAssignment

Return type

RetentionAssignment

```
assignable_policy(reference: str, status: bool)
```

Make a policy assignable

Parameters

- **reference** (*str*) – The policy ID
- **status** (*bool*) – The assignable status

Returns

```
assignments(entity: Entity) → Set[RetentionAssignment]
```

Return a list of retention policies for an entity.

Parameters

entity (*class:Entity*) – The entity to fetch assignments for

Returns

Set of policy assignments

Return type

Set[*RetentionAssignment*]

```
create_policy(**kwargs)
```

Create a new policy

Arguments are kwargs map

Name Description SecurityTag StartDateField Period PeriodUnit ExpiryAction ExpiryActionParameters
Restriction Assignable

```
delete_policy(reference: str)
```

Delete a retention policy

Parameters

reference (*str*) – The policy reference

```
policies() → Set[RetentionPolicy]
```

Return a list of all retention policies Only returns the first 250 policies in the system

Returns

Set of retention policies

Return type

Set[*RetentionPolicy*]

policy(*reference: str*) → *RetentionPolicy*

Return a retention policy by reference

Parameters

reference (*str*) – The policy reference

Returns

The retention policy

Return type

RetentionPolicy

policy_by_name(*name: str*) → *RetentionPolicy*

Return a retention policy by name

Parameters

name (*str*) – The policy name

Returns

The retention policy

Return type

RetentionPolicy

remove_assignments(*retention_assignment: RetentionAssignment*)

Delete a retention policy from an asset

Parameters

retention_assignment (*RetentionAssignment*) – The Preservica Entity to assign a policy to

Returns

The Asset Reference

Return type

str

update_policy(*reference: str, **kwargs*)

Update an existing policy

Arguments are kwargs map

Name Description SecurityTag StartDateField Period PeriodUnit ExpiryAction ExpiryActionParameters
Restriction Assignable

24.5 Workflow API

Note: The Workflow API is available for Enterprise Preservica users

<https://eu.preservica.com/api/admin/documentation.html>

class pyPreservica.**WorkflowContext**(*workflow_id, workflow_name: str*)

Defines a workflow context. The workflow context is the pre-defined workflow which is ready to run

class pyPreservica.**WorkflowInstance**(instance_id: int)

Defines a workflow Instance. The workflow Instance is context which has been executed

class pyPreservica.**WorkflowAPI**(username: str | None = None, password: str | None = None, tenant: str | None = None, server: str | None = None, use_shared_secret: bool = False, two_fa_secret_key: str | None = None, protocol: str = 'https')

A class for calling the Preservica Workflow API

This API can be used to programmatically manage the Preservica Workflows.

<https://preview.preservica.com/sdb/rest/workflow/documentation.html>

get_workflow_contexts(definition: str)

Return a list of Workflow Contexts which have the same Workflow Definition

Parameters

definition (str) – The Workflow Definition ID

Returns

List of Workflow Contexts

Return type

list

get_workflow_contexts_by_type(workflow_type: str)

Return a list of Workflow Contexts which have the same Workflow type

Parameters

workflow_type (str) – The Workflow type Ingest, Access, Transformation or DataManagement

Returns

List of Workflow Contexts

Return type

list

start_workflow_instance(workflow_context: WorkflowContext, **kwargs)

Start a workflow context

Returns a Correlation Id which is used to monitor the workflow progress

Parameters

- **workflow_context** (WorkflowContext) – The workflow context to start
- **kwargs** – Key/Values to pass to the workflow instance

Returns

correlation_id

Return type

str

terminate_workflow_instance(instance_ids)

Terminate a workflow by its instance id

Parameters

instance_ids (int or a list of int) – The Workflow instance

workflow_instance(*instance_id: int*) → *WorkflowInstance*

Return a workflow instance by its Id

Parameters

instance_id (*int*) – The Workflow instance

Returns

workflow_instance

Return type

WorkflowInstance

workflow_instances(*workflow_state: str, workflow_type: str, **kwargs*)

Return a list of Workflow instances

Parameters

- **workflow_state** – The Workflow state Aborted, Active, Completed, Finished_Mixed_Outcome, Pending, Suspended, Unknown, or Failed
- **workflow_type** – The Workflow type Ingest, Access, Transformation or DataManagement

24.6 Administration and Management API

Note: The Administration and Management API needs to be enabled by the help desk.

<https://eu.preservica.com/sdb/rest/workflow/documentation.html>

```
class pyPreservica.AdminAPI(username: str | None = None, password: str | None = None, tenant: str | None = None, server: str | None = None, use_shared_secret: bool = False, two_fa_secret_key: str | None = None, protocol: str = 'https')
```

add_security_tag(*tag_name*) → str

Create a new security tag

Parameters

tag_name (*str*) – The new security tag

Returns

The new security tag

Return type

str

add_system_role(*role_name*) → str

Create a new user roles

Parameters

role_name (*str*) – The new role

Returns

The new role

Return type

str

add_user(*username: str, full_name: str, roles: list, externally_authenticated: bool = False*)

Add a new user

Parameters

- **externally_authenticated**
- **username** (*str*) – email address of the preservica user
- **full_name** (*str*) – Users real name
- **roles** (*list*) – List of roles assigned to the user

Returns

dictionary of user attributes

Return type

dict

add_xml_document(*name: str, xml_data: Any, document_type: str = 'MetadataTemplate'*)

Add a new XML document to Preservica The default type of XML document is a descriptive metadata template

Options are:

MetadataDropdownLists -> Authority Lists CustomIndexDefinition -> Custom Search Indexes MetadataTemplate -> Metadata Template UploadWizardConfigurationFile -> Upload Wizard Config ConfigurationFile -> Heritrix Config File

Parameters

- **name** (*str*) – The name of the xml document
- **xml_data** – The xml schema as a UTF-8 string or as a file like object
- **document_type** (*str*) – The type of the XML document, defaults to descriptive metadata templates

Returns

None

Return type

None

add_xml_schema(*name: str, description: str, originalName: str, xml_data: Any*)

Add a new XSD document to Preservica

Parameters

- **name** (*str*) – Name for the XSD schema
- **description** (*str*) – Description for the XSD schema
- **originalName** (*str*) – The original file name for the schema on disk
- **xml_data** (*Any*) – The xml schema as a UTF-8 string or a file like object

Returns

None

Return type

None

add_xml_transform(*name: str, input_uri: str, output_uri: str, purpose: str, originalName: str, xml_data: Any*)

Add a new XML transform to Preservica

Parameters

- **name** (*str*) – The name of the XML transform
- **input_uri** (*str*) – The URI of the input XML document
- **output_uri** (*str*) – The URI of the output XML document
- **purpose** (*str*) – The purpose of the transform, “transform”, “edit”, “view”
- **originalName** (*str*) – The original file name of the transform
- **xml_data** (*Any*) – The transform xml as a string or file like object

Returns

None

Return type

None

all_users() → list

Return a list of all users in the system

Return list of usernames

Return type

list

change_user_display_name(*username: str, new_display_name: str*) → dict

Change the user display name

Parameters

- **username** (*str*) – email address of the preservica user
- **new_display_name** (*str*) – Users real name

Returns

dictionary of user attributes

Return type

dict

delete_security_tag(*tag_name*)

Delete a security tag

Parameters

- **tag_name** (*str*) – The security tag to delete

delete_system_role(*role_name*)

Delete a system role

Parameters

- **role_name** (*str*) – The role to delete

delete_user(*username: str*)

Delete a user

Parameters

- **username** (*str*) – email address of the preservica user

delete_xml_document(*uri: str*)

Delete a XML document from Preservica

Parameters

uri (*str*) – The URI of the xml document to delete

Returns

None

Return type

None

delete_xml_schema(*uri: str*)

Delete an XML schema from Preservica

Parameters

uri (*str*) – The URI of the xml schema to delete

Returns

None

Return type

None

delete_xml_transform(*input_uri: str, output_uri: str*)

Delete a XSD document from Preservica

Parameters

- **input_uri** (*str*) – The URI of the input XML document
- **output_uri** (*str*) – The URI of the output XML document

Returns

None

Return type

None

disable_user(*username*)

Disable a Preservica User to prevent them logging in

Parameters

username (*str*) – email address of the preservica user

enable_user(*username*)

Enable a Preservica User

Parameters

username (*str*) – email address of the preservica user

security_tags() → list

List all the security tags in the system

Returns

list of security tags

Return type

list

system_roles() → list

List all the user access roles in the system

Returns

list of roles

Return type

list

user_details(*username: str*) → dict

Get the details of a user by their username

Parameters

username (*str*) – email address of the preservica user

Returns

dictionary of user attributes

Return type

dict

user_report(*report_name='users.csv'*)

Create a report on all tenancy users :return:

xml_document(*uri: str*) → str

fetch the metadata XML document as a string by its URI

Parameters

uri (*str*) – The URI of the xml document

Returns

The XML document as a string

Return type

str

xml_documents() → List

fetch the list of XML documents stored in Preservica

Returns

List of XML documents stored in Preservica

Return type

list

xml_schema(*uri: str*) → str

fetch the metadata schema XSD document as a string by its URI

Parameters

uri (*str*) – The URI of the xml schema

Returns

The XML schema as a string

Return type

str

xml_schemas() → List

fetch the list of metadata schema XSD documents stored in Preservica

Returns

List of XML schema's stored in Preservica

Return type

list

xml_transform(*input_uri: str, output_uri: str*) → str

fetch the XML transform as a string by its URIs

Parameters

- **input_uri** (*str*) – The URI of the input XML document
- **output_uri** (*str*) – The URI of the output XML document

Returns

The XML transform as a string

Return type

str

xml_transforms() → List

fetch the list of xml transforms stored in Preservica

Returns

List of XML transforms stored in Preservica

Return type

list

24.7 Process Monitor API

<https://us.preservica.com/api/processmonitor/documentation.html>

class pyPreservica.**MonitorStatus**(*value*)

An enumeration.

class pyPreservica.**MonitorCategory**(*value*)

An enumeration.

class pyPreservica.**MonitorAPI**(*username: str | None = None, password: str | None = None, tenant: str | None = None, server: str | None = None, use_shared_secret: bool = False, two_fa_secret_key: str | None = None, protocol: str = 'https'*)

A class for the Preservica Repository Process Monitor API

<https://us.preservica.com/api/processmonitor/documentation.html>

API for retrieving and updating monitoring information about processes.

messages(*monitor_id, status: MessageStatus | None = None*) → Generator

List of messages for a process

Parameters

- **monitor_id** (*str*) – The Process ID
- **status** (*MessageStatus*) – The message status, info, warning, error etc

Returns

Generator for each message, each message is a dict object

monitors(*status*: [MonitorStatus](#) | *None* = *None*, *category*: [MonitorCategory](#) | *None* = *None*) → Generator

Get a filtered list of non-abandoned process monitors

Parameters

- **status** ([MonitorStatus](#)) – process status values (Pending, Running, Succeeded, Failed, Suspended, Recoverable)
- **category** ([MonitorCategory](#)) – process categories (Ingest, Export, DataManagement, Automated)

Returns

Generator for each monitor

timeseries(*monitor_id*)

Get the historical record of progress for a single monitor

Parameters

monitor_id (*str*) – The Process ID

Returns

List of timeseries information

24.8 WebHook API

<https://us.preservica.com/api/webhook/documentation.html>

class pyPreservica.**TriggerType**(*value*)

Enumeration of the Web hooks Trigger Types

class pyPreservica.**WebHooksAPI**(*username*: *str* | *None* = *None*, *password*: *str* | *None* = *None*, *tenant*: *str* | *None* = *None*, *server*: *str* | *None* = *None*, *use_shared_secret*: *bool* = *False*, *two_fa_secret_key*: *str* | *None* = *None*, *protocol*: *str* = 'https')

Class to register new webhook endpoints

subscribe(*url*: *str*, *triggerType*: [TriggerType](#), *secret*: *str*)

Subscribe to a new web hook

Parameters

- **url**
- **triggerType**
- **secret**

Returns

json_response

subscriptions()

Return all the current active web hook subscriptions as a json document

Returns

list of web hooks

unsubscribe(*subscription_id*: *str*)

Unsubscribe from the provided webhook.

Parameters

subscription_id

Returns**unsubscribe_all()**

Unsubscribe from all webhooks. :return:

24.9 Authority Records API

<https://eu.preservica.com/api/reference-metadata/documentation.html>

This API is used for managing the Authority records within Preservica.

```
class pyPreservica.Table(reference: str, name: str, security_tag: str, displayField: str, metadataConnections: list)
```

```
class pyPreservica.AuthorityAPI(username: str | None = None, password: str | None = None, tenant: str | None = None, server: str | None = None, use_shared_secret: bool = False, two_fa_secret_key: str | None = None, protocol: str = 'https')
```

```
add_record(table: Table, record: dict)
```

Add a new record to an existing table

Parameters

- **table** – The Table to add the record to
- **record** – The record

Type

table: Table

Type

record: dict

Returns

A single record

Return type

dict

```
add_records(table: Table, csv_file, encoding=None)
```

Add new records to an existing table from a CSV document

Parameters

- **table** – The Table to add the record to
- **csv_file** – The path to the CSV document
- **encoding** – The encoding used to open the csv document

Type

table: Table

Type

csv_file: str

Type

encoding: str

delete_record(*reference: str*)

Delete a record from a table by its reference

Parameters

reference – The reference of the record to delete

Type

reference: str

record(*reference: str*) → dict

Return a record by its reference

Parameters

reference – The record reference

Type

reference: str

Returns

A single record

Return type

dict

records(*table: Table*) → List[dict]

Return all records from a table

Parameters

table – The authority table

Type

table: Table

Returns

List of records

Return type

list[dict]

table(*reference: str*) → *Table*

fetch an authority table by its reference

Parameters

reference – The reference for the authority table

Type

reference: str

Returns

An authority table

Return type

Table

tables() → Set[*Table*]

List reference metadata tables

Returns

Set of authority tables

Return type

set(*Table*)

INDEX

- `genindex`

PYTHON MODULE INDEX

p

`pyPreservica`, [1](#)

A

active (*pyPreservica.Generation* attribute), 107
 add_assignments() (*pyPreservica.RetentionAPI* method), 116
 add_identifier() (*pyPreservica.EntityAPI* method), 100
 add_metadata() (*pyPreservica.EntityAPI* method), 101
 add_record() (*pyPreservica.AuthorityAPI* method), 126
 add_records() (*pyPreservica.AuthorityAPI* method), 126
 add_security_tag() (*pyPreservica.AdminAPI* method), 119
 add_system_role() (*pyPreservica.AdminAPI* method), 119
 add_thumbnail() (*pyPreservica.EntityAPI* method), 105
 add_user() (*pyPreservica.AdminAPI* method), 119
 add_xml_document() (*pyPreservica.AdminAPI* method), 120
 add_xml_schema() (*pyPreservica.AdminAPI* method), 120
 add_xml_transform() (*pyPreservica.AdminAPI* method), 120
 AdminAPI (class in *pyPreservica*), 119
 all_descendants() (*pyPreservica.EntityAPI* method), 103
 all_events() (*pyPreservica.EntityAPI* method), 105
 all_users() (*pyPreservica.AdminAPI* method), 121
 Asset (class in *pyPreservica*), 108
 asset (*pyPreservica.Representation* attribute), 107
 asset() (*pyPreservica.EntityAPI* method), 97
 assignable_policy() (*pyPreservica.RetentionAPI* method), 116
 assignments() (*pyPreservica.RetentionAPI* method), 116
 AuthorityAPI (class in *pyPreservica*), 126

B

Bitstream (class in *pyPreservica*), 107
 bitstream_content() (*pyPreservica.EntityAPI* method), 100

bitstreams (*pyPreservica.Generation* attribute), 107

C

change_user_display_name() (*pyPreservica.AdminAPI* method), 121
 children() (*pyPreservica.EntityAPI* method), 103
 complex_asset_package() (in module *pyPreservica*), 112
 content_object() (*pyPreservica.EntityAPI* method), 97
 content_objects() (*pyPreservica.EntityAPI* method), 99
 ContentAPI (class in *pyPreservica*), 110
 ContentObject (class in *pyPreservica*), 109
 create_folder() (*pyPreservica.EntityAPI* method), 99
 create_policy() (*pyPreservica.RetentionAPI* method), 116
 cvs_to_xml() (in module *pyPreservica*), 113

D

delete_asset() (*pyPreservica.EntityAPI* method), 103
 delete_folder() (*pyPreservica.EntityAPI* method), 104
 delete_identifiers() (*pyPreservica.EntityAPI* method), 100
 delete_metadata() (*pyPreservica.EntityAPI* method), 102
 delete_policy() (*pyPreservica.RetentionAPI* method), 116
 delete_record() (*pyPreservica.AuthorityAPI* method), 126
 delete_security_tag() (*pyPreservica.AdminAPI* method), 121
 delete_system_role() (*pyPreservica.AdminAPI* method), 121
 delete_user() (*pyPreservica.AdminAPI* method), 121
 delete_xml_document() (*pyPreservica.AdminAPI* method), 121
 delete_xml_schema() (*pyPreservica.AdminAPI* method), 122
 delete_xml_transform() (*pyPreservica.AdminAPI* method), 122

descendants() (*pyPreservica.EntityAPI* method), 103
description (*pyPreservica.Asset* attribute), 108
description (*pyPreservica.ContentObject* attribute), 109
description (*pyPreservica.Entity* attribute), 108
description (*pyPreservica.Folder* attribute), 109
disable_user() (*pyPreservica.AdminAPI* method), 122
download() (*pyPreservica.EntityAPI* method), 104

E

effective_date (*pyPreservica.Generation* attribute), 107
enable_user() (*pyPreservica.AdminAPI* method), 122
Entity (class in *pyPreservica*), 107
entity() (*pyPreservica.EntityAPI* method), 98
entity_events() (*pyPreservica.EntityAPI* method), 105
entity_type (*pyPreservica.Asset* attribute), 109
entity_type (*pyPreservica.ContentObject* attribute), 110
entity_type (*pyPreservica.Entity* attribute), 108
entity_type (*pyPreservica.Folder* attribute), 109
EntityAPI (class in *pyPreservica*), 97
EntityType (class in *pyPreservica*), 110
export_opex() (*pyPreservica.EntityAPI* method), 106

F

filename (*pyPreservica.Bitstream* attribute), 107
fixity (*pyPreservica.Bitstream* attribute), 107
Folder (class in *pyPreservica*), 109
folder() (*pyPreservica.EntityAPI* method), 97
format_group (*pyPreservica.Generation* attribute), 107
formats (*pyPreservica.Generation* attribute), 107

G

Generation (class in *pyPreservica*), 106
generations() (*pyPreservica.EntityAPI* method), 99
get_async_progress() (*pyPreservica.EntityAPI* method), 106
get_workflow_contexts() (*pyPreservica.WorkflowAPI* method), 118
get_workflow_contexts_by_type() (*pyPreservica.WorkflowAPI* method), 118

I

identifier() (*pyPreservica.EntityAPI* method), 100
identifiers_for_entity() (*pyPreservica.EntityAPI* method), 100
indexed_fields() (*pyPreservica.ContentAPI* method), 110
ingest_tweet() (*pyPreservica.UploadAPI* method), 113
ingest_twitter_feed() (*pyPreservica.UploadAPI* method), 113

ingest_web_video() (*pyPreservica.UploadAPI* method), 114
IntegrityCheck (class in *pyPreservica*), 110

L

length (*pyPreservica.Bitstream* attribute), 107

M

messages() (*pyPreservica.MonitorAPI* method), 124
metadata (*pyPreservica.Asset* attribute), 108
metadata (*pyPreservica.ContentObject* attribute), 110
metadata (*pyPreservica.Entity* attribute), 108
metadata (*pyPreservica.Folder* attribute), 109
metadata() (*pyPreservica.EntityAPI* method), 101
metadata_for_entity() (*pyPreservica.EntityAPI* method), 101
module
 pyPreservica, 1
MonitorAPI (class in *pyPreservica*), 124
MonitorCategory (class in *pyPreservica*), 124
monitors() (*pyPreservica.MonitorAPI* method), 124
MonitorStatus (class in *pyPreservica*), 124
move() (*pyPreservica.EntityAPI* method), 102
move_async() (*pyPreservica.EntityAPI* method), 102
move_sync() (*pyPreservica.EntityAPI* method), 102

N

name (*pyPreservica.Representation* attribute), 107

O

object_details() (*pyPreservica.ContentAPI* method), 110
original (*pyPreservica.Generation* attribute), 106

P

parent (*pyPreservica.Asset* attribute), 108
parent (*pyPreservica.ContentObject* attribute), 109
parent (*pyPreservica.Entity* attribute), 108
parent (*pyPreservica.Folder* attribute), 109
policies() (*pyPreservica.RetentionAPI* method), 116
policy() (*pyPreservica.RetentionAPI* method), 117
policy_by_name() (*pyPreservica.RetentionAPI* method), 117
properties (*pyPreservica.Generation* attribute), 107
pyPreservica
 module, 1

R

record() (*pyPreservica.AuthorityAPI* method), 127
records() (*pyPreservica.AuthorityAPI* method), 127
reference (*pyPreservica.Asset* attribute), 108
reference (*pyPreservica.ContentObject* attribute), 109
reference (*pyPreservica.Entity* attribute), 107

reference (*pyPreservica.Folder* attribute), 109
 RelationshipDirection (*class in pyPreservica*), 110
 remove_assignments() (*pyPreservica.RetentionAPI* method), 117
 remove_thumbnail() (*pyPreservica.EntityAPI* method), 105
 rep_type (*pyPreservica.Representation* attribute), 107
 replace_generation_async() (*pyPreservica.EntityAPI* method), 106
 replace_generation_sync() (*pyPreservica.EntityAPI* method), 105
 Representation (*class in pyPreservica*), 107
 representations() (*pyPreservica.EntityAPI* method), 99
 RetentionAPI (*class in pyPreservica*), 115
 RetentionAssignment (*class in pyPreservica*), 115
 RetentionPolicy (*class in pyPreservica*), 115

S

save() (*pyPreservica.EntityAPI* method), 98
 security_tag (*pyPreservica.Asset* attribute), 108
 security_tag (*pyPreservica.ContentObject* attribute), 109
 security_tag (*pyPreservica.Entity* attribute), 108
 security_tag (*pyPreservica.Folder* attribute), 109
 security_tag_async() (*pyPreservica.EntityAPI* method), 98
 security_tag_sync() (*pyPreservica.EntityAPI* method), 98
 security_tags() (*pyPreservica.AdminAPI* method), 122
 simple_asset_package() (*in module pyPreservica*), 111
 simple_search_csv() (*pyPreservica.ContentAPI* method), 111
 simple_search_list() (*pyPreservica.ContentAPI* method), 110
 start_workflow_instance() (*pyPreservica.WorkflowAPI* method), 118
 subscribe() (*pyPreservica.WebHooksAPI* method), 125
 subscriptions() (*pyPreservica.WebHooksAPI* method), 125
 system_roles() (*pyPreservica.AdminAPI* method), 122

T

Table (*class in pyPreservica*), 126
 table() (*pyPreservica.AuthorityAPI* method), 127
 tables() (*pyPreservica.AuthorityAPI* method), 127
 terminate_workflow_instance() (*pyPreservica.WorkflowAPI* method), 118
 thumbnail() (*pyPreservica.EntityAPI* method), 104
 timeseries() (*pyPreservica.MonitorAPI* method), 125
 title (*pyPreservica.Asset* attribute), 108

title (*pyPreservica.ContentObject* attribute), 109
 title (*pyPreservica.Entity* attribute), 108
 title (*pyPreservica.Folder* attribute), 109
 TriggerType (*class in pyPreservica*), 125

U

unsubscribe() (*pyPreservica.WebHooksAPI* method), 125
 unsubscribe_all() (*pyPreservica.WebHooksAPI* method), 126
 update_metadata() (*pyPreservica.EntityAPI* method), 101
 update_policy() (*pyPreservica.RetentionAPI* method), 117
 updated_entities() (*pyPreservica.EntityAPI* method), 104
 upload_buckets() (*pyPreservica.UploadAPI* method), 114
 upload_credentials() (*pyPreservica.UploadAPI* method), 114
 upload_locations() (*pyPreservica.UploadAPI* method), 114
 upload_zip_package() (*pyPreservica.UploadAPI* method), 114
 upload_zip_package_to_Azure() (*pyPreservica.UploadAPI* method), 114
 upload_zip_package_to_S3() (*pyPreservica.UploadAPI* method), 115
 upload_zip_to_Source() (*pyPreservica.UploadAPI* method), 115
 UploadAPI (*class in pyPreservica*), 113
 user_details() (*pyPreservica.AdminAPI* method), 123
 user_report() (*pyPreservica.AdminAPI* method), 123

W

WebHooksAPI (*class in pyPreservica*), 125
 workflow_instance() (*pyPreservica.WorkflowAPI* method), 118
 workflow_instances() (*pyPreservica.WorkflowAPI* method), 119
 WorkflowAPI (*class in pyPreservica*), 118
 WorkflowContext (*class in pyPreservica*), 117
 WorkflowInstance (*class in pyPreservica*), 117

X

xml_document() (*pyPreservica.AdminAPI* method), 123
 xml_documents() (*pyPreservica.AdminAPI* method), 123
 xml_schema() (*pyPreservica.AdminAPI* method), 123
 xml_schemas() (*pyPreservica.AdminAPI* method), 123
 xml_transform() (*pyPreservica.AdminAPI* method), 124
 xml_transforms() (*pyPreservica.AdminAPI* method), 124