pyPreservica Documentation

Release v6.1

Sep 22, 2020

Contents

pyPreservica is python library for the Preservica API

This library provides a Python class for working with the Preservica Entity Rest API

This version of the library is compatible with Preservica versions 6.0 and 6.1

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

Table of Contents

CHAPTER 1

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 metadata 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.

CHAPTER 2

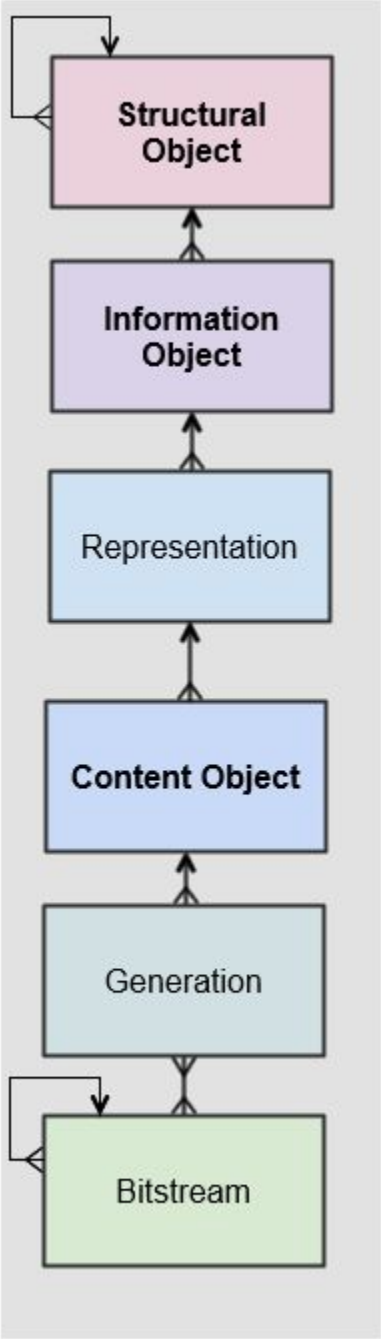
Features

- 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
- Fetch Folders and Assets belonging to parent Folders
- Move Assets and Folders within the repository
- Retrieve Representations, Generations & Bitstreams from Assets
- Download digital files and thumbnails
- Fetch lists of changed entities over the last n days.

CHAPTER 3

Background

They 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.

CHAPTER 4

PIP Installation

pyPreservica is available from the Python Package Index (PyPI)

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

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
```

CHAPTER 5

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
```

CHAPTER 6

Contributing

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

CHAPTER 7

Example

Create the entity API client object and request an asset (information object) by its unique identifier

```
>>> from pyPreservica import *
>>> client = EntityAPI()
>>> asset = client.asset("9bad5acf-e7a1-458a-927d-2d1e7f15974d")
>>> print(asset.title)
```

CHAPTER 8

Authentication

pyPreservica provides 3 different methods for authentication. The library requires the username and password of a Preservica user and a Tenant identifier along with the server hostname.

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 then 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" and save to the working directory

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

(continues on next page)

(continued from previous page)

```
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()
```

CHAPTER 9

SSL Certificates

pyPreservica will only connect to servers which use the https:// protocol and will always validate certificates.

pyPreservica uses the Certifi project to provide SSL certificate validation.

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

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
```

CHAPTER 10

The User Guide

10.1 Entity API QuickStart

Making a call to the Preservica repository is very simple.

Begin by importing the pyPreservica module

```
>>> from pyPreservica import *
```

Now, let's create the EntityAPI class

```
>>> client = EntityAPI()
```

10.1.1 Fetching Entities (Assets, Folders & Content Objects)

Fetch an asset and print its attributes

```
>>> 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)
```

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 the parent objects of an asset all the way to the root of the repository

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

10.1.2 Fetching Children of Entities

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

To get a set of all the root folders use

```
>>> root_folders = client.children(None)
```

or

```
>>> root_folders = client.children()
```

To get a set of children of a particular folder use

```
>>> entities = client.children(folder.reference)
```

To get the siblings of an asset you can use

```
>>> entities = client.children(asset.parent)
```

The set of entities returned may contain both assets and other folders. The default size of the result set is 50 items. The size can be configured and for large result sets paging is available.

```
>>> next_page = None
>>> while True:
>>>     root_folders = client.children(None, maximum=10, next_page=next_page)
>>>     for e in root_folders.results:
>>>         print(f'{e.title} : {e.reference} : {e.entity_type}')
>>>         if not root_folders.has_more:
```

(continues on next page)

(continued from previous page)

```
>>>             break
>>>         else:
>>>             next_page = root_folders.next_page
```

A version of this method is also available as a generator function which does not require explicit paging. This version returns a lazy iterator which does the paging internally. It will default to 50 items between server requests

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

You can pass a parent reference to get the children of any folder in the same way as the explict paging version

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

This is the preferred way to get children of folders as the paging is managed automatically.

If you only need the folders or assets from a parent you can filter the results using a pre-defined filter

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

or

```
>>> 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 the you can call all_descendants() this is a generator function which returns a lazy iterator will which 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)
```

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)
```

10.1.3 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
```

10.1.4 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)
```

To change the security tag on an Asset or Folder we have a separate API. Since this may be a long running process for folders containing many assets you can choose either a asynchronous (non-blocking) or synchronous (blocking call)

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

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

The synchronous version will block until the security tag has been updated on all entities.

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

10.1.5 3rd Party External Identifiers

We can add external identifiers to either assets, folders or content objects. External identifiers have a type and a value. External identifiers do not have to be unique in the same way as internal identifiers.

```
>>> 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 e in client.identifier("DOI", "urn:nbn:de:1111-20091210269"):
>>>     o = client.entity(e.entity_type, e.reference)
>>>     print(o.title)
>>>     print(o.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="122334
↪")
```

Will only delete identifiers which match the type and value

10.1.6 Descriptive Metadata

You can query an entity to determine if it has any attached descriptive metadata using the metadata attribute. This returns a dict[] object the dictionary key is a url to the metadata and the value is the schema

```
>>> 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 to the metadata() method

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

An alternative is to call the metadata_for_entity method directly

```
>>> xml_document = client.metadata_for_entity(entity, "https://www.person.com/person")
```

to fetch the first metadata template which matches the schema argument on the entity

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://www.person.com/person'>" \
        "<person:Name>Bob Smith</person:Name>" \
```

(continues on next page)

(continued from previous page)

```
        "<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://www.person.com/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)
```

Descriptive metadata can also be updated to amend values or change the document structure

```
>>> folder = entity.folder("723f6f27-c894-4ce0-8e58-4c15a526330e")    # call into the␣
↪API
>>>
>>> for url, schema in folder.metadata.items():
>>>     if schema == "https://www.person.com/person":
>>>         xml_string = entity.metadata(url)                        # call into the API
>>>         xml_document = ElementTree.fromstring(xml_string)
>>>         postcode = ElementTree.Element('{https://www.person.com/person}Postcode')
>>>         postcode.text = "OX14 3YS"
>>>         xml_document.append(postcode)
>>>         xml_string = ElementTree.tostring(xml_document, encoding='UTF-8', xml_
↪declaration=True).decode("utf-8")
>>>         entity.update_metadata(folder, schema, xml_string)    # call into the API
```

10.1.7 Representations, Content Objects & Generations

Each asset in Preservica contains one or more representations, such as Preservation or Access etc.

To get a list of all the representations of an asset

```
>>> 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 whereas more complex objects such as 3D models, books, multi-page documents may have several content objects.

```
>>> for content_object in client.content_objects(representation):
>>>     print(content_object.reference)
>>>     print(content_object.title)
>>>     print(content_object.description)
>>>     print(content_object.parent)
>>>     print(content_object.metadata)
>>>     print(content_object.asset.title)
```

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)
```

(continues on next page)

(continued from previous page)

```
>>>     print(generation.content_object)
>>>     print(generation.format_group)
>>>     print(generation.effective_date)
>>>     print(generation.bitstreams)
```

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

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

The actual content files can be download using bitstream_content()

```
>>> client.bitstream_content(bs, bs.filename)
```

10.1.8 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.

```
>>> client.move(folder, None)
```

10.1.9 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.

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.

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.jpg")
```

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.jpg", Thumbnail.LARGE)      ##␣
↪400×400   pixels
>>> filename = client.thumbnail(asset, "thumbnail.jpg", Thumbnail.MEDIUM)     ##␣
↪150×150   pixels
>>> filename = client.thumbnail(asset, "thumbnail.jpg", Thumbnail.SMALL)      ## 64×64␣
↪    pixels
```

10.2 Content API QuickStart

pyPreservica now contains some experimental interfaces to the content API

https://demo.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.

Warning: Unlike the entity API above the interfaces for the content API are subject to change.

The content API client is created using

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

10.2.1 object-details

Get the details for a Asset or Folder as a raw json document:

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

10.2.2 indexed-fields

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

```
>>> client = ContentAPI()
>>> client.indexed_fields():
```

10.2.3 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("%", "results.csv")

>>> client = ContentAPI()
>>> client.simple_search_csv("Oxford", "oxford.csv")

>>> client = ContentAPI()
>>> 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)
```

10.3 Entity API Developer Interface

This part of the documentation covers all the interfaces of pyPreservica.

All of the pyPreservica functionality can be accessed by these methods on the *EntityAPI* object.

class pyPreservica.**EntityAPI**

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

Returns The entity
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 process ID
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
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
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 type list(*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 type set(*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 A 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 document 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(*entity*, *dest_folder*)

Move an entity (asset or folder) to a new folder

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_reference*, *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_reference** (*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_reference*)

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_reference** (*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_reference*)

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 50 elements. Callers can iterate over the result to get all children with a single call.

param str folder_reference The parent folder reference, None for the children of root folders

return A set of entity objects (Folders and Assets)

rtype set(Entity)

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

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

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

Map 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 map of descriptive metadata attached to the asset.

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

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

Assets have entity type EntityType.CONTENT_OBJECT

10.4 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)
```

(continues on next page)

(continued from previous page)

```
            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-
↪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"
```

(continues on next page)

(continued from previous page)

```
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)
            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
```

(continues on next page)

(continued from previous page)

```
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/"})
    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
```

Python Module Index

p

Index

N

O

P

R

S

T

U