aiomixcloud Documentation

Release 1.0.6

Author

Oct 11, 2020

Contents:

CHAPTER 1

Mixcloud API wrapper for Python and Async IO

aiomixcloud is a wrapper library for the HTTP API of Mixcloud. It supports asynchronous operation via asyncio and specifically the aiohttp framework. *aiomixcloud* tries to be abstract and independent of the API's transient structure, meaning it is not tied to specific JSON fields and resource types. That is, when the API changes or expands, the library should be ready to handle it.

1.1 Usage

1.1.1 Basic usage

The library's main interface is the *Mixcloud* class. Import it directly from `aiomixcloud` and use it as an asynchronous context manager. Pass a key (URL part corresponding to a unique API resource) to its *get()* method to fetch information about that *resource*:

```
from aiomixcloud import Mixcloud

async with Mixcloud() as mixcloud:
    user = await mixcloud.get('bob')
```

Result data is available both as attributes and dictionary items:

```
user.city  # 'London'
user['favorite_count']  # 38
user.pictures['medium']  # 'https://thumbnailer.mixcloud.com/unsafe/10...'
```

Datetime data gets automatically converted to `datetime` objects:

```
user.updated_time  # datetime.datetime(2018, 3, 10, 7, 32, tzinfo=tzutc())
```

The *discover()* shortcut returns information about a tag:

```
tag = await mixcloud.discover('jazz')
tag['name']  # 'Jazz shows'
```

Resource objects are dict-like and can be iterated over:

```
# Using `tag` from previous snippet
for key, value in tag.items():
    print(key, value)
```

Original `dict` is stored in their *data* attribute:

```
tag.data  # {'url': 'https://www.mixcloud.com/discover/jazz/', 'type': ...
```

1.1.2 Listing

The listing methods consist of:

- *popular()*
- *hot()*
- *new()*
- *search()*

and are responsible for downloading a list of resources:

```
popular = await mixcloud.popular()
popular  # <ResourceList 'Popular Cloudcasts'>

# Index the list
popular[0]  # <Resource: Cloudcast '/alice/pop-hits-episode-42/'>

# Items are resources
popular[3]['audio_length']  # 1676

# Iterate
for p in popular:
    p.url  # 'https://www.mixcloud.com/...'
```

Items of a resource list can also be accessed by their key:

```
popular['chris/funky-mix']  # <Resource: Cloudcast '/chris/funky-mix/'>
```

The *search()* method, which can accept a *type* argument, among `'cloudcast'` (default), `'user'` and `'tag'`, returns resources matching the given term:

```
rock_cloudcasts = await mixcloud.search('rock')
johns = await mixcloud.search('john', type='user')
```

Listing methods can accept pagination arguments the API itself defines: *offset*, *limit*, *since* and *until*. The former two concern net numbers (counts) and the latter two can be UNIX timestamps, human-readable strings or `datetime` objects. Alternatively, instead any of those, a *page* argument can be specified (zero-indexed), giving 20 results per page (unless the *per_page* argument indicates otherwise):

```
hot = await mixcloud.hot(offset=40, limit=80)
new = await mixcloud.new(since='2018 Feb 12 13:00:00',
                         until='2019 March 28 21:15:04')
some_jazz = await mixcloud.search('jazz', page=2)
metal_music = await mixcloud.search('metal', page=4, per_page=30)
```

Note: Timezone-naive datetime values (either human-readable strings or `datetime` *objects) will be treated as being in the current local timezone.*

Resource lists have a *previous()* and a *next()* method which return the previous and the next page of the current resource list, respecitvely. If there is no such page available, these methods return `None`:

```
older_metal = await metal_music.previous()
older_metal  # <ResourceList>
newer_metal = await metal_music.next()
newer_metal  # <ResourceList>
await newer_metal.next()  # This returns None
```

When responding with a *resource list*, the API sends most of the information for each resource, but not all of it. That is an example of dealing with *non-full* resources. Again, in a resource list, some of the data included, represent resources related to each list item, for example each item in a cloudcast list contains information about the user who uploaded the cloudcast. The information about that user is also incomplete, making it another case of a non-full resource. The *load()* method of *Resource* objects can be used to load the full information of a non-full resource:

```
# Using `hot` from previous snippet
some_hot_cloudcast = hot[5]
some_hot_cloudcast.description  # raises AttributeError
await some_hot_cloudcast.load()
some_hot_cloudcast.description  # 'The greatest set of all time...'
```

load() also returns the freshly-loaded object so it can be used in chained calls, something that can find elegant application in *synchronous library usage*.

1.1.3 Connections

API resources can have sub-resources, or, *connections*, that is other API resources associated with (or, "owned" by) them. For example, a user can have followers, i.e a user resource has *followers* as a connection, which are other user resources themselves. The connections of a resource become available through methods of it, named after the respective connection names:

```
peter = await mixcloud.get('peter')
his_followers = await peter.followers()
his_followers  # <ResourceList "Peter's followers">

nice_cloudcast = await mixcloud.get('luke/a-nice-mix')
comments = await nice_cloudcast.comments()
for comment in comments:
    comment  # <Resource: Comment '/comments/cr/.../'>
    comment.comment  # 'Nice set, keep up the good work!'
```

1.1.4 Embedding

Embedding information and HTML code for a cloudcast can be retrieved through the *embed_json()* and *embed_html()* methods, being able to take *width*, *height* and *color* as arguments:

```
json_embed_info = await mixcloud.embed_json('someuser/the-best-mix')
html_embed_code = await mixcloud.embed_html('someuser/the-best-mix',
                                            width=300, height=150)
```

oEmbed information for a resource (previous arguments applicable here as well) is available through:

```
oembed_info = await mixcloud.oembed(resource_key)
```

1.1.5 Authorization

Significant part of the API's functionality is available after OAuth authorization. Acquiring an OAuth access token to enable authorized, personalized API calls, requires obtaining a "client ID" and a "client secret". This can be done by applying to Mixcloud to "create an application". As authorization is currently allowed only through web browser, the user must be redirected to a URL ("authorization URL") where they will be able to "allow the application access to their data". This URL must contain a "client ID" and a "redirect URI" as GET parameters. Once the user allows access, they will be redirected to that developer-chosen "redirect URI" with a *code* ("OAuth code") GET parameter. Finally, with a request to an appropriate URL, the developer can exchange this OAuth code with an access token they can use on behalf of the user.

The *MixcloudOAuth* class assists the process of acquiring an OAuth access token:

```
from aiomixcloud.auth import MixcloudOAuth

oauth = MixcloudOAuth(client_id=CLIENT_ID, client_secret=CLIENT_SECRET,
                      redirect_uri='https://example.com/store-code')
oauth.authorization_url  # Forward user here to prompt them to allow
                         # access to your application
```

Once the user allows access to your application they will be redirected to *https://example.com/store_code?code=OAUTH_CODE* and you can use the passed *code* GET parameter to get their access token:

```
access_token = await oauth.access_token(code)
async with Mixcloud(access_token=access_token) as mixcloud:
    # Authorized use of the API here
    pass
```

This process can, alternatively, take place after the instantiation of the *Mixcloud* class, to make use of its session:

```
async with Mixcloud() as mixcloud:
    oauth = MixcloudOAuth(client_id=CLIENT_ID,
                          client_secret=CLIENT_SECRET,
                          redirect_uri='https://example.com/store-code',
                          mixcloud=mixcloud)
    # ... After getting user's permission and storing `code` ...
    mixcloud.access_token = await oauth.access_token(code)
```

Apart from getting richer results from some of the API calls, authorized usage enables access to personalized methods, concerning the user who the access token corresponds to. The simplest of them is *me()*, which gives the resource of the access token owner *(current user)*:

```
current_user = await mixcloud.me()
current_user.username  # 'amikrop'
```

1.1.6 Actions

Authorized usage also enables *actions*, a group of methods about doing and undoing certain actions on specific resources:

follow()	*unfollow()*
favorite()	*unfavorite()*
repost()	*unrepost()*
listen_later()	*unlisten_later()*

Each of them takes a resource key as an argument (the two methods on the first row target a user, the rest of them target a cloudcast):

```
data = await mixcloud.follow('bob')
data['result']['message']  # 'Now following bob'
data = await mixcloud.unrepost('alice/fun-times-ep-25')
data.result.success  # True
```

1.1.7 Uploading

Making authorized use of the API allows uploading cloudcasts and editing existing uploads. Both *upload()* and *edit()* share the following optional arguments: *picture* (filename), *description* (text), *tags* (sequence of strings), sections (sequence of mappings) and some fields available only to pro accounts: *publish_date* (UNIX timestamp, human-readable string or `datetime` object), *disable_comments* (boolean), *hide_stats* (boolean) and *unlisted* (boolean).

The *upload()* method takes two positional arguments, *mp3* (filename) and *name* (string):

```
data = await mixcloud.upload('perfectmix.mp3', 'Perfect Mix',
                             picture='perfectpic.jpg',
                             description='The perfect house mix',
                             tags=['house', 'deep'],
                             sections=[{'chapter': 'Intro',
                                        'start_time': 0},
                                       {'artist': 'Somebody',
                                        'song': 'Some song',
                                        'start_time': 60},
                                       {'artist': 'Cool DJ',
                                        'song': 'Cool track',
                                        'start_time': 240}])
data.result['success']  # True
```

edit() takes a *key* positional argument and a *name* optional argument:

```
data = await mixcloud.edit('amikrop/perfect-mix', name='The Perfect Mix',
                           description='The best house mix, right for summer',
                           tags=['house', 'deep', 'summer'])
data['result'].success  # True
```

1.1.8 Targeting

Methods of *Mixcloud* that target a specific resource (and thus, take a key as first argument) are also available as methods of *Resource* objects:

```
someone = await mixcloud.get('certainuser')
await someone.unfollow()  # {'result': ...

mix = await mixcloud.get('auser/acloudcast')
await mix.favorite()  # {'result': ...
```

(continues on next page)

(continued from previous page)

```
await mix.embed_html()  # '<iframe width="100%" height=...'

my_mix = await mixcloud.get('amikrop/perfect-mix')
await my_mix.edit(description='The best house mix, perfect for summer!',
                  tags=['house', 'deep',
                        'summer', 'smooth'])  # {'result': ...
```

Those methods include the *actions*, the embedding methods and `edit()`.

1.1.9 Synchronous mode

All the functionality of the package is also available for synchronous (i.e blocking) usage. `MixcloudSync` and `MixcloudOAuthSync` provide the same interface as their asynchronous versions, with all the coroutine methods now being classic methods. Context management becomes synchronous and methods of returned objects are synchronous as well:

```
from aiomixcloud.sync import MixcloudOAuthSync, MixcloudSync

with MixcloudSync() as mixcloud:
    oauth = MixcloudOAuthSync(client_id=CLIENT_ID,
                              client_secret=CLIENT_SECRET,
                              redirect_uri=REDIRECT_URI,
                              mixcloud=mixcloud)
    # ... After getting user's permission and storing `code` ...
    mixcloud.access_token = oauth.access_token(code)

    some_cloudcast = mixcloud.get('someuser/somemix')
    some_cloudcast.repost()  # {'result': ...

    # Chained calls
    some_cloudcast.similar()[0].load().picture_primary_color  # '02f102'
```

1.2 Module Reference

1.2.1 aiomixcloud.auth module

1.2.1.1 API access authorization

This module contains the class for Mixcloud API OAuth authorization. Specifically:

- `MixcloudOAuth`, producing authorization URLs and trading OAuth codes for access tokens.

class aiomixcloud.auth.**MixcloudOAuth**(*oauth_root='https://www.mixcloud.com/oauth'*, ***, *client_id=None*, *client_secret=None*, *redirect_uri=None*, *raise_exceptions=None*, *mixcloud=None*)

Bases: `object`

Mixcloud OAuth authorization

By having `client_id` and `redirect_uri` set, a `MixcloudOAuth` object provides `authorization_url`, a URL to forward the end user to, where they will be able to "allow the ap-

plication access to their data". It also provides the *access_token()* method, which trades an OAuth code for an access token (requiring *client_secret* as well as the previously mentionted attributes).

_build_url(*segment*)
Return a URL consisting of *OAuth root*, followed by *segment*.

_check()
Check that *client ID* and *redirect URI* have been set.

_oauth_root = None
Base URL for OAuth-related requests

_raise_exceptions = None
Whether to raise an exception when API responds with an error message. If not specified, use the respective setting of the *mixcloud* attribute. If that attribute is not specified, default to False.

access_token(*code*)
Send OAuth *code* to server and get an access token. If fail raise *MixcloudOAuthError* in case this is the setting (*self._raise_exceptions* or self.mixcloud._raise_exceptions), otherwise return None.

authorization_url
Return authorization URL.

client_id = None
Client ID, provided by Mixcloud

client_secret = None
Client secret, provided by Mixcloud

mixcloud = None
The *Mixcloud* object whose session will be used to make the request from. If None, a new session will be created for the access token request.

oauth_root = 'https://www.mixcloud.com/oauth'
Default Mixcloud OAuth root URL

redirect_uri = None
Redirect URI, chosen by the developer

1.2.2 aiomixcloud.constants module

1.2.2.1 Constant values

This module includes constant, hard-coded values used throughout the package.

aiomixcloud.constants.**API_ROOT = 'https://api.mixcloud.com'**
Mixcloud API root URL

aiomixcloud.constants.**DESCRIPTION_MAX_SIZE = 1000**
Uploaded description maximum allowed size, in characters

aiomixcloud.constants.**MIXCLOUD_ROOT = 'https://www.mixcloud.com'**
Mixcloud root URL

aiomixcloud.constants.**MP3_MAX_SIZE = 4294967296**
Uploaded mp3 maximum allowed size, in bytes

aiomixcloud.constants.**OAUTH_ROOT = 'https://www.mixcloud.com/oauth'**
Mixcloud OAuth root URL

aiomixcloud.constants.**OEMBED_ROOT = 'https://www.mixcloud.com/oembed'**
Mixcloud oEmbed root URL

aiomixcloud.constants.**PICTURE_MAX_SIZE = 10485760**
Uploaded picture maximum allowed size, in bytes

aiomixcloud.constants.**TAG_MAX_COUNT = 5**
Tag maximum allowed count per upload

1.2.3 aiomixcloud.core module

1.2.3.1 Main functionality coordination

This module contains the class responsible for aggregating and organizing the main functionality and usage of the package. Specifically:

- *Mixcloud*, providing the interface of the package. Stores the basic configuration and preferences, holds the session which API calls are made from and has all the methods necessary to reach every endpoint of the API, as well as take advantage of its capabilities.

class aiomixcloud.core.**Mixcloud**(*api_root='https://api.mixcloud.com'*, ***, *access_token=None*, *mixcloud_root='https://www.mixcloud.com'*, *oembed_root='https://www.mixcloud.com/oembed'*, *json_decoder_class=<class 'aiomixcloud.json.MixcloudJSONDecoder'>*, *resource_class=<class 'aiomixcloud.models.Resource'>*, *resource_list_class=<class 'aiomixcloud.models.ResourceList'>*, *raise_exceptions=False*, *session=None*)

Bases: object

Asynchronous Mixcloud API handler

This class orchestrates the interaction of the package's user with the Mixcloud API. Being ready for use with no explicit configuration at all, as well as being capable of customized operation, it provides the methods to hit the API. A common to use method and base for many of the rest, is the *get()* method, which takes a "key", a URL segment corresponding to a unique API resource and returns a native, yet familiar and handy data structure representing that resource. There, also, exist a family of methods, like *popular()*, which receive pagination arguments and return a list of resources. The class provides some personalized methods, that is methods which require authorization to run, through an access token. These are methods like *follow()* and *unfollow()*. In this category belong the methods about uploading data to the server, *upload()* and *edit()*. Finally, there are methods about getting embedding information for some resource, like *embed_html()*. The class can be used as an asynchronous context manager to avoid having to call *close()* explicitly, which closes the session.

_api_root = None
Base URL for all API requests

_build_url(*segment*)
Return a URL consisting of API root, followed by *segment*.

_delete_action(*url*, *action*)
Make HTTP DELETE request about *action*, to resource specified by *url* and return results.

_do_action(*url*, *action*, *method*)
Make HTTP *method* request about *action*, to resource specified by *url* and return results.

_embed(*cloudcast*, *params*)
Get embed data for *cloudcast*, in desirable format using specified display options.

_json_decode = None
JSON decode function

_mixcloud_root = None
Base Mixcloud URL

_native_result(*response*)
Wrap *_process_response()* and return appropriate *AccessDict* object.

_oembed_root = None
Base URL for all oEmbed requests

_post_action(*url*, *action*)
Make HTTP POST request about *action*, to resource specified by *url* and return results.

_process_response(*response*)
Return JSON-decoded data out of *response*. If fail to decode, let JSONDecodeError pass through in case *_raise_exceptions* is set, otherwise return None.

_proper_result(*response*)
Return the proper kind of result, based on Content-Type of *response*.

_raise_exceptions = None
Whether to raise an exception when API responds with error message

_resource_class = None
Class for resource model

_resource_list_class = None
Class for resource model list

_session = None
The ClientSession object to make requests from

_upload(*params*, *data*, *url*)
Add multipart fields from *params* to possibly half-filled *data*, POST it to *url* and return results.

static _url_join(*url*, *segment*)
Return a URL consisting of *url*, followed by *segment*. Strip possibly existing leading slash of *segment*, for joining to work.

access_token = None
OAuth Access token

api_root = 'https://api.mixcloud.com'
Default Mixcloud API root URL

close()
Close *_session*.

discover(*tag*)
Get information about *tag*.

edit(*key*, *params*, *, *name=None*)
Edit upload identified by *key*, as described by specified parameters.

embed_html(**args*, ***kwargs*)
Get embed data for given cloudcast, in HTML format using specified display options.

embed_json(**args*, ***kwargs*)
Get embed data for given cloudcast, in JSON format using specified display options.

favorite(*cloudcast*)
Favorite *cloudcast* and return results of the request.

follow(*user*)
Follow *user* and return results of the request.

get(*url*, *, *relative=True*, *create_connections=True*, ***params*)
Send a GET request to API and return JSON-decoded data. If *relative* is `True`, *url* is considered relative to the API root, otherwise it is considered as an absolute URL.

hot(*params*)
Get information about hot cloudcasts.

json_decoder_class
alias of *aiomixcloud.json.MixcloudJSONDecoder*

listen_later(*cloudcast*)
Add *cloudcast* to "listen later" list and return results of the request.

me()
Get information about user authorized by used access token.

mixcloud_root = 'https://www.mixcloud.com'
Default Mixcloud root URL

new(*params*)
Get information about new cloudcasts.

oembed(*key*, *params*)
Get oEmbed data for resource identified by *key*, in desirable format using specified display options.

oembed_root = 'https://www.mixcloud.com/oembed'
Default Mixcloud oEmbed root URL

popular(*params*)
Get information about popular cloudcasts.

repost(*cloudcast*)
Repost *cloudcast* and return results of the request.

resource_class
alias of *aiomixcloud.models.Resource*

resource_list_class
alias of *aiomixcloud.models.ResourceList*

search(*query*, *params*, *, *type='cloudcast'*)
Search resources of *type* by *query* and return found information.

unfavorite(*cloudcast*)
Unfavorite *cloudcast* and return results of the request.

unfollow(*user*)
Unfollow *user* and return results of the request.

unlisten_later(*cloudcast*)
Remove *cloudcast* from "listen later" list and return results of the request.

unrepost(*cloudcast*)
Unrepost *cloudcast* and return results of the request.

upload(*mp3*, *name*, *params*)
Upload file with filename indicated by *mp3*, named *name* and described by specified parameters.

1.2.4 aiomixcloud.datetime module

1.2.4.1 Datetime formatting

This module contains functions about presenting datetimes in various formats. Specifically:

- `format_datetime()`, returning a "YYYY-MM-DDTHH:MM:SSZ"-formatted datetime string out of a datetime-like value.
- `to_timestamp()`, returning a UNIX timestamp out of a datetime-like value.

The datetime-like argument of those functions can be a `datetime` object, a human-readable datetime string or a timestamp. Timezone-naive values will be treated as being in the local timezone. Result will be converted in UTC, if not already there.

aiomixcloud.datetime.**_as_utc**(*dt*)
Convert and return given *dt* in UTC. If it is timezone-naive, treat it as being in the local timezone.

aiomixcloud.datetime.**_to_datetime**(*value*)
Return a `datetime` object out of a datetime-like *value*. Raise `TypeError` on invalid argument.

aiomixcloud.datetime.**format_datetime**(*value*)
Return a datetime string in the "YYYY-MM-DDTHH:MM:SSZ" format, out of a datetime-like *value*.

aiomixcloud.datetime.**to_timestamp**(*value*)
Return a UNIX timestamp out of a datetime-like *value*.

1.2.5 aiomixcloud.decorators module

1.2.5.1 Function decorators

This module contains decorators for functions of the package. Specifically:

- `displayed()`, handling requests with varying response format and HTML display information by defining the proper arguments and passing a *dict* produced out of them to the decorated coroutine.
- `paginated()`, handling pagination of resource lists by defining the proper arguments, checking their validity and passing a *dict* produced out of them to the decorated coroutine.
- `personal()`, checking that *access_token* is set on the object which owns the decorated coroutine method, before awaiting it.
- `targeting()`, marking the decorated coroutine as one that targets a specific resource identified by a key, making said coroutine available as a `Resource` method.
- `uploading()`, handling requests that upload data to the server by defining the proper arguments and passing a *dict* produced out of them to the decorated coroutine.

aiomixcloud.decorators.**displayed**(*coroutine*)
Return a coroutine that takes arguments about desired response format and HTML display and makes a parameters dictionary out of them. Then, it passes that dictionary to the original coroutine, while awaiting it.

aiomixcloud.decorators.**paginated**(*coroutine*)
Return a coroutine that takes pagination arguments, runs checks on them and makes a parameters dictionary out of them. Then, it passes that dictionary to the original coroutine, while awaiting it.

aiomixcloud.decorators.**personal**(*coroutine*)
Return a coroutine method that checks if *access_token* is set on *self*.

aiomixcloud.decorators.**targeting**(*coroutine*)
Mark *coroutine* as one that targets a specific resource, so it can be used as a `Resource` method.

aiomixcloud.decorators.**uploading**(*coroutine*)
Return a coroutine that takes arguments about uploading cloudcast-related data, runs checks on them and makes a parameters dictionary out of them. Then, it passes that dictionary to the original coroutine, while awaiting it.

1.2.6 aiomixcloud.exceptions module

1.2.6.1 Exception types

This module contains custom exceptions raised during use of the package. Specifically:

- *MixcloudError*, indicating that API returned an error response.
- *MixcloudOAuthError*, indicating that something went wrong with the OAuth authorization process.

exception aiomixcloud.exceptions.**MixcloudError**(*data*)
Bases: Exception

Raised when API responds with error message.

extra = None
Dictionary with additional information

message = None
Error message

type = None
Error type, as described by the API

exception aiomixcloud.exceptions.**MixcloudOAuthError**(*data*)
Bases: Exception

Raised when OAuth authorization fails.

message = None
Error message

1.2.7 aiomixcloud.json module

1.2.7.1 JSON decoder

This module contains the class responsible for decoding the JSON data retrieved from the API requests. Specifically:

- *MixcloudJSONDecoder*, turning datetime-like strings to datetime objects.

class aiomixcloud.json.**MixcloudJSONDecoder**(**args*, ***kwargs*)
Bases: json.decoder.JSONDecoder

Handles datetime values.

static object_hook(*obj*)
Turn eligible values to datetime objects and leave the rest unchanged.

1.2.8 aiomixcloud.models module

1.2.8.1 Basic data structures

This module contains the basic data structures used throughout the package. Specifically:

- *_WrapMixin*, a mixin providing type casting of accessed items.
- *AccessDict*, dict-like, supports accessing items by attribute. Accessed items are properly wrapped.
- *AccessList*, list-like, supports accessing *Resource* items by their "key" key. Accessed items are properly wrapped.
- *Resource*, like an *AccessDict* which has a "type" key. Gets methods for downloading its "connections", that is the sub-entities "owned" by the resource. Mirrors methods of *Mixcloud* marked as "targeted" with their first argument as its key. Also provides a *Resource.load()* method to download full resource information in case it is partly loaded as an element of another API response.
- *ResourceList*, like an *AccessDict* which delegates string indexing and iterating over, to its "data" item. Supports pagination through the *ResourceList.previous()* and *ResourceList.next()* methods, which return a new object with the respective data.

class aiomixcloud.models.**AccessDict**(*data*, *, *mixcloud*)
Bases: *aiomixcloud.models._WrapMixin*, collections.UserDict

Dict-like model which supports accessing items using keys as attributes. Items are wrapped with a proper model, depending on their type. Original *dict* is stored in *self.data*.

_abc_impl = <_abc_data object>

class aiomixcloud.models.**AccessList**(*data*, *, *mixcloud*)
Bases: *aiomixcloud.models._WrapMixin*, collections.UserList

List-like model which supports accessing *Resource*-like items by matching their "key" item. Items are wrapped with a proper model, depending on their type. Original *list* is stored in *self.data*.

_abc_impl = <_abc_data object>

class aiomixcloud.models.**Resource**(*data*, *, *full=False*, *create_connections=True*, *mixcloud*)
Bases: *aiomixcloud.models.AccessDict*

Mixcloud API resource

A resource is like an *AccessDict* object which has a "type" key. When a "type" key is present in an API (sub)object, suggesting it has a unique URL, it is considered an API resource, that is an individual entity (a user, a cloudcast, a tag, etc). A *Resource* object has appropriately named methods for downloading information about its sub-entities ("connections"). It also mirrors "targeted" methods of its *Mixcloud* instance, passing them its key as a first argument. Targeted methods include "actions" (e.g *follow()* or *unfavorite()*), embed-related methods and *edit()*.

_abc_impl = <_abc_data object>

_create_connections()
In case there is an item with a `['metadata']['connections']` key, create a method for each of its items (resource "connections") that fetches information about sub-entities associated with the resource (eg *comments*, *followers* etc). Each of these methods is named after the respective connection.

_full = None
Whether all of resource data has been downloaded (by having accessed the detail page).

load(*, *force=False*)
Load full resource information from detail page. Do nothing in case *_full* is `True`, unless *force* is set. Return *self*, so this can be used in chained calls.

class aiomixcloud.models.**ResourceList**(*data*, *, *mixcloud*)
Bases: *aiomixcloud.models.AccessDict*

Contains a list of resources, with paging capabilities. Main data is stored in the `'data'` item, while a `'paging'` item may be present indicating URLs of the previous and next pages of the resource list, as well as

a *'name'* item describing the collection. Indexing falls back to `self['data']` on failure, while iterating over and length concern "data" straight up.

_abc_impl = <_abc_data object>

_navigate(*where*)
Return an adjacent page of current resource list (another *ResourceList* object) specified by *where*, or `None` if it is not found.

next()
Return next page of current resource list, or `None` if it is not found.

previous()
Return previous page of current resource list, or `None` if it is not found.

class aiomixcloud.models.**_WrapMixin**(*data*, *, *mixcloud*)
Bases: object

Enables returning the proper kind of object, when indexed and iterated over. Produces aiomixcloud models out of dictionaries and lists, leaving the rest of the types intact.

_wrap(*item*)
Wrap *item* with proper class. *dict* becomes *AccessDict*, or *Resource*-like if it has `"type"` as a key. *list* becomes *AccessList*. The rest remain unchanged.

mixcloud = None
`Mixcloud` instance to pass along to contained items

1.2.9 aiomixcloud.sync module

1.2.9.1 Synchronous operation mode

This module contains synchronous (i.e blocking) versions of the package classes. Specifically:

- *MixcloudOAuthSync*, synchronous version of *MixcloudOAuth*, handling OAuth authorization.
- *MixcloudSync*, synchronous version of *Mixcloud*, handling main functionality coordination.

class aiomixcloud.sync.**MixcloudOAuthSync**(**args*, ***kwargs*)
Bases: object

Synchronous version of *MixcloudOAuth*.

method(**args*, *method_name='__repr__'*)
Mirror original object's method with given *method_name*.

method_name = '__repr__'

name = 'repr'

class aiomixcloud.sync.**MixcloudSync**(**args*, ***kwargs*)
Bases: *aiomixcloud.sync.MixcloudSync*

Synchronous version of *Mixcloud* with synchronous context management capabilities.

class aiomixcloud.sync.**ResourceListSync**(**args*, ***kwargs*)
Bases: object

Synchronous version of *ResourceList*.

method(**args*, *method_name='__repr__'*)
Mirror original object's method with given *method_name*.

method_name = '__repr__'

name = 'repr'

class aiomixcloud.sync.**ResourceSync**(**args*, ***kwargs*)
Bases: object

Synchronous version of *Resource*.

method(**args*, *method_name='__repr__'*)
Mirror original object's method with given *method_name*.

method_name = '__repr__'

name = 'repr'

aiomixcloud.sync.**_MixcloudSync**
Synchronous version of *Mixcloud* without synchronous context management capabilities.

alias of *aiomixcloud.sync.MixcloudSync*

aiomixcloud.sync.**_make_sync**(*cls*, ***options*)
Return a synchronous version of *cls*, freezing *options* as keyword arguments to its constructor.

CHAPTER 2

Indices and tables

- genindex
- modindex

Python Module Index

a

Index

Symbols

A

R

S

T

U