clarifai Documentation

Release 2.0.26

Clarifai

Jun 13, 2017

First steps

CHAPTER 1

First steps

Installation Guide

Note: Generally, the python client works with Python 2.x and Python 3.x. But it is only tested against 2.7 and 3.5. Feel free to report BUGs if you encounter on other versions.

Install the package

You can install the latest stable clarifai using pip (which is the canonical way to install Python packages). To install using `pip` run:

```
pip install clarifai==2.0.26
```

You can also install from git using latest source:

```
pip install git+git://github.com/Clarifai/clarifai-python.git
```

Configuration

The client uses CLARIFAI_APP_ID and CLARIFAI_APP_SECRET for authentication and token generation. Each application you create uses its own unique ID and secret to authenticate requests. The client will use the authentication information passed to it by three methods with the following precedence order:: + Passed in to the constructor through the *app_id* and *app_secret* parameters. + Set as the CLARIFAI_APP_ID and CLARIFAI_APP_SECRET environment variables + Placed in the .clarifai/config file using the command below. You can get these values from https://developer.clarifai.com/account/applications and then run:

```
$ clarifai config
CLARIFAI_APP_ID: []: **********************************YQEd
CLARIFAI_APP_SECRET: []: **********************************gCqT
```

If you do not see any error message after this, you are all set and can proceed with using the client.

Windows Users

For Windows users, you may fail running the `clarifai config` when you try to configure the runtime environment. This is because Windows uses file extension to determine executables and by default file `clarifai` without file extension is nonexecutables. In order to run the command, you may want to launch it with the python interpreter.

```
C:\Python27>python.exe Scripts\clarifai config
CLARIFAI_APP_ID: []: **********************************YQEd
CLARIFAI_APP_SECRET: []: **********************************gCqT
```

AWS Lambda Users

For AWS Lambda users, in order to use the library correctly, you are recommended to set two environmental variables *CLARIFAI_APP_ID* and *CLARIFAI_APP_SECRET* in the lambda function configuration, or hardcode the APP_ID and APP_SECRET in the API instantiation.

Tutorial

Each of the examples below is a small independent code snippet within 10 lines that could work by copy and paste to a python source code file. By playing with them, you should be getting started with Clarifai API. For more usages about the API, check the API Reference.

Predict with general model

```
from clarifai.rest import ClarifaiApp

app = ClarifaiApp()

app.tag_urls(['https://samples.clarifai.com/metro-north.jpg'])
```

Predict with travel model

```
from clarifai.rest import ClarifaiApp

app = ClarifaiApp()

app.tag_urls(urls=['https://samples.clarifai.com/wedding.jpg'], model='travel-v1.0')
```

Predict with food model

```
from clarifai.rest import ClarifaiApp

app = ClarifaiApp()

app.tag_urls(urls=['https://samples.clarifai.com/wedding.jpg'], model='food-items-v1.0
↪')
```

Predict with NSFW model

```
from clarifai.rest import ClarifaiApp

app = ClarifaiApp()

app.tag_urls(urls=['https://samples.clarifai.com/wedding.jpg'], model='nsfw-v1.0')
```

Predict with color model

```
from clarifai.rest import ClarifaiApp

app = ClarifaiApp()

model = app.models.get('color', model_type='color')

image2 = ClImage(url='https://samples.clarifai.com/wedding.jpg')

model.predict([image1, image2])
```

Upload Images

```
from clarifai.rest import ClarifaiApp

app = ClarifaiApp()

app.inputs.create_image_from_url(url='https://samples.clarifai.com/puppy.jpeg',␣
↪concepts=['my puppy'])
app.inputs.create_image_from_url(url='https://samples.clarifai.com/wedding.jpg', not_
↪concepts=['my puppy'])
```

Create a Model

Note: This assumes you follow through the tutorial and finished the "Upload Images" Otherwise you may not be able to create the model.

```
model = app.models.create(model_id="puppy", concepts=["my puppy"])
```

Train the Model

Note: This assumes you follow through the tutorial and finished the "Upload Images" and "Create a Model" to create a model. Otherwise you may not be able to train the model.

```
model.train()
```

Predict with Model

Note:

This assumes you follow through the tutorial and finished the "Upload Images", "Create a Model", and "Train the Model".

Otherwise you may not be able to make predictions with the model.

```
from clarifai.rest import ClarifaiApp

app = ClarifaiApp()

model = app.models.get('puppy')
model.predict_by_url('https://samples.clarifai.com/metro-north.jpg')
```

Instantiate an Image

```
from clarifai.rest import Image as ClImage

# make an image with an url
img = ClImage(url='https://samples.clarifai.com/dog1.jpeg')

# make an image with a filename
img = ClImage(filename='/tmp/user/dog.jpg')

# allow duplicate url
img = ClImage(url='https://samples.clarifai.com/dog1.jpeg', allow_dup_url=True)

# make an image with concepts
img = ClImage(url='https://samples.clarifai.com/dog1.jpeg', \
              concepts=['cat', 'animal'])

# make an image with metadata
img = ClImage(url='https://samples.clarifai.com/dog1.jpeg', \
              concepts=['cat', 'animal'], \
              metadata={'id':123,
                        'city':'New York'
                       })
```

Bulk Import Images

If you have a large amount of images, you may not want to upload them one by one by calling *app.inputs.create_image_from_url('https://samples.clarifai.com/dog1.jpeg')*

Instead you may want to use the bulk import API.

Note: The max number images per batch is 128. If you have more than 128 images to upload, you may want to chunk them into 128 or less, and bulk import them batch by batch.

In order to use this, you have to instantiate Image() objects from various sources.

```
from clarifai.rest import ClarifaiApp
from clarifai.rest import Image as ClImage

# assume there are 100 urls in the list
images = []
for url in urls:
  img = ClImage(url=url)
  images.append(img)

app.inputs.bulk_create_images(images)
```

Search the Image

Note: This assumes you follow through the tutorial and finished the "Upload Images" Otherwise you may not be able to search

```
from clarifai.rest import ClarifaiApp

app = ClarifaiApp()

app.inputs.search_by_annotated_concepts(concept='my puppy')

app.inputs.search_by_predicted_concepts(concept='dog')

app.inputs.search_by_image(url='https://samples.clarifai.com/dog1.jpeg')

app.inputs.search_by_metadata(metadata={'key':'value'})
```

Basic Concepts

This page lists a few basic concepts used in the Clarifai API.

Image

Image is straightforward. It represents a picture in digital format.

In Clarifai API, an image could be represented by an url, a local filename, an opened io stream, raw bytes of the image, or bytes encoded in base64.

There is a Image() class to construct an Image object in Clarifai API.

Along with the image bytes, an image could also be associated with an unique ID, as well as one or more concepts indicating what it is and what is it not.

Image object could be used in uploading, prediction, as well as search.

Input

Input is a generalized image, because input could be image, or video in the near future. In the current API, we use Image and Input interchangably because Image is the only Input type the API supports.

Similarly, input could be associated with ID and concepts. And Input is used in uploading, prediction, and search.

Concept

Concept represents a class to associate with images. It could be a concrete concept like "dog" or "cat", or a abstract concept like "love". It is the output of the concept Models. Clarifai Concept has an unique ID, and a name. Sometimes the concept name is also referred to label or tag for the predictions. They are exchangably used somewhere.

Model

Model is machine learning algorithm that takes input, such as image or video, and output some results for prediction. For custom training, the models trained are all concept models. The output of a concept model is a list of concepts in prediction, associated with probabilities for condidence of the prediction.

Image Crop

Image crop is a the definition of the crop box within an image. We use this in visual search so user does not have to crop an image before the search.

We use percentage coordinates instead of pixel coordinates to specify the crop box.

A four-element-tuple represents a crop box, in (top_y, left_x, bottom_y, right_x) order.

So a (0.3, 0.2, 0.6, 0.4) represents a box horizontally spanning from 20%-40% and vertically spanning 30%-60% of the image.

Installation Guide Install Clarifai python library

Tutorial Getting started with Clarifai API using python client

Basic Concepts Getting familiar with basic concepts used in Clarifai API

API Reference

This is the API Reference documentation extracted from the source code.

Application

class `clarifai.rest.client.`**`ClarifaiApp`**(*app_id=None*, *app_secret=None*, *base_url=None*, *api_key=None*, *quiet=True*)

Clarifai Application Object

This is the entry point of the Clarifai Client API With authentication to an application, you can access all the models, concepts, inputs in this application through the attributes of this class.

To access the models: use app.models To access the inputs: use app.inputs To access the concepts: use app.concepts

check_upgrade()
check client upgrade if the client has been installed for more than one week, the check will be triggered. If the newer version is available, a prompt message will be poped up as a warning message in STDERR. The API call will not be paused or interrupted.

tag_files(*files*, *model='general-v1.3'*)

tag files on disk with user specified models by default tagged by 'general-v1.3' model

Parameters

- **files** – a list of local file names for tagging the max lens of the list is 128, which is the max batch size
- **model** – the model name to tag with default model is general model for general tagging purpose

Returns the JSON string from the predict call

Examples

```
>>> files = ['/tmp/metro-north.jpg',        >>>           '/tmp/dog2.jpeg']
>>> app.tag_urls(files)
```

tag_urls(*urls*, *model='general-v1.3'*)

tag urls with user specified models by default tagged by 'general-v1.3' model

Parameters

- **urls** – a list of URLs for tagging the max lens of the list is 128, which is the max batch size
- **model** – the model name to tag with default model is general model for general tagging purpose

Returns the JSON string from the predict call

Examples

```
>>> urls = ['https://samples.clarifai.com/metro-north.jpg',        >>>
↪'https://samples.clarifai.com/dog2.jpeg']
>>> app.tag_urls(urls)
```

wait_until_inputs_delete_finish()
block until the inputs deletion finishes

The criteria of inputs deletion finish is 0 inputs from GET /inputs

Parameters **void** –

Returns void

wait_until_models_delete_finish()
block until the inputs deletion finishes

The criteria of models deletion finish is 0 private models from GET /models

Parameters **void** –

Returns void

Concepts

class clarifai.rest.client.**Concepts**(*api*)

bulk_create(*concept_ids*, *concept_names=None*)
bulk create concepts

When the concept name is not set, it will be set as the same as concept ID.

Parameters

- **concept_ids** – a list of concept IDs
- **concept_names** – a list of concept name

Returns A list of Concept() object

Examples::

```
>>> app.inputs.bulk_create(['id1', 'id2'], ['cute cat', 'cute dog'])
```

bulk_update(*concept_ids*, *concept_names*, *action='overwrite'*)
patch multiple concepts

Parameters

- **concept_ids** – a list of concept_id, in sequence
- **concept_names** – a list of corresponding concept names, in the same sequence

Returns the new concept object

Examples

```
>>> app.concepts.bulk_update(concept_ids=['myid1', 'myid2'], concept_names=[
↪'name2', 'name3'])
```

create(*concept_id*, *concept_name=None*)
create a new concept

Parameters

- **concept_id** – concept ID, the unique identifier of the concept
- **concept_name** – name of the concept If name is not specified, it will be set to the same as concept ID

Returns the new Concept object

get(*concept_id*)
get a concept by id

Parameters **concept_id** – concept ID, the unique identifier of the concept

Returns If found, return the Concept object Otherwise, return None

Examples

```
>>> app.concepts.get('id')
```

get_all()
get all concepts in a generator

Parameters **void** –

Returns all concepts in a generator

get_by_page(*page=1*, *per_page=20*)
get concept with pagination

Parameters

- **page** – page number
- **per_page** – number of inputs to retrieve per page

Returns a list of Concept object

Examples

```
>>> for concept in app.concepts.get_by_page(2, 10):
>>>   print concept.concept_id
```

search(*term*, *lang=None*)
search concepts by concept name with wildcards

Parameters

- **term** – search term with wildcards
- **lang** – language to search

Returns a list concept in a generator

Examples

```
>>> app.concepts.search('cat')
>>> # search for Chinese label name
>>> app.concepts.search(u'*')
```

update(*concept_id*, *concept_name*, *action='overwrite'*)
patch concept

Parameters

- **concept_id** – id of the concept
- **concept_name** – name of the concept that you want to change to

Returns the new concept object

Examples

```
>>> app.concepts.update(concept_id='myid1', concept_name='new_concept_name2')
```

Inputs

class clarifai.rest.client.**Inputs**(*api*)

add_concepts(*input_id*, *concepts*, *not_concepts*)
add concepts for one input

This is just an alias of *merge_concepts* for easier understanding when you try to add some new concepts to an image

Parameters

- **input_id** – the unique ID of the input
- **concepts** – the list of concepts
- **not_concepts** – the list of negative concepts

Returns an Input object

Examples

```
>>> app.inputs.add_concepts('id', ['cat', 'kitty'], ['dog'])
```

bulk_create_images(*images*)
bulk create images

Parameters **images** – a list of Image object

Returns a list of Image object just got created

Examples

```
>>> img1 = Image(url="", concepts=['cat', 'kitty'])
>>> img2 = Image(url="", concepts=['dog'], not_concepts=['cat'])
>>> app.inputs.bulk_create_images([img1, img2])
```

bulk_delete_concepts(*input_ids*, *concept_lists*)
bulk delete concepts from a list of input ids

Parameters

- **input_ids** – a list of input IDs
- **concept_lists** – a list of concept list

Returns an Input object

Examples

```
>>> app.inputs.bulk_delete_concepts(['id'], [['cat', 'dog']])
```

bulk_merge_concepts(*input_ids*, *concept_lists*)
bulk merge concepts from a list of input ids

Parameters

- **input_ids** – a list of input IDs
- **concept_lists** – a list of concept list

Returns an Input object

Examples

```
>>> app.inputs.bulk_merge_concepts('id', [[('cat',True), ('dog',False)]])
```

bulk_update(*images*, *action='merge'*)
update the input update the information of an input/image

Parameters

- **images** – a list of Image() objects that have concepts, metadata, etc.
- **method** – one of ['merge', 'overwrite'] 'merge' is to merge the info into the exising info, for either concept or metadata 'overwrite' is to overwrite the metadata, concepts with the existing ones

Returns an Image object

Examples

```
>>> new_img1 = Image(image_id="abc1", concepts=['c1', 'c2'], not_concepts=['c3
↪'], metadata={'key':'val'})
>>> new_img2 = Image(image_id="abc2", concepts=['c1', 'c2'], not_concepts=['c3
↪'], metadata={'key':'val'})
>>> app.inputs.update([new_img1, new_img2], action='overwrite')
```

check_status()
check the input upload status

Parameters **Void** –

Returns InputCounts object

Examples

```
>>> status = app.inputs.check_status()
>>> print status.code
>>> print status.description
```

create_image(*image*)
create an image from Image object

Parameters **image** – a Clarifai Image object

Returns the image object just got created and uploaded

Examples::

```
>>> app.inputs.create_image(Image(url='https://samples.clarifai.com/metro-
↪north.jpg'))
```

create_image_from_base64(*base64_bytes*, *image_id=None*, *concepts=None*, *not_concepts=None*, *crop=None*, *metadata=None*, *geo=None*, *allow_duplicate_url=False*)

create an image by base64 bytes

Parameters

- **base64_bytes** – base64 encoded image bytes
- **image_id** – ID of the image
- **concepts** – a list of concepts
- **not_concepts** – a list of concepts
- **crop** – crop information, with four corner coordinates
- **metadata** – meta data with a dictionary
- **geo** – geo info with a dictionary
- **allow_duplicate_url** – True of False, the flag to allow duplicate url to be imported

Returns the image object just got created and uploaded

Examples::

```
>>> app.inputs.create_image_bytes(base64_bytes="base64 encoded image␣
↪bytes...")
```

create_image_from_bytes(*img_bytes*, *image_id=None*, *concepts=None*, *not_concepts=None*, *crop=None*, *metadata=None*, *geo=None*, *allow_duplicate_url=False*)

create an image by image bytes

Parameters

- **img_bytes** – raw bytes of an image
- **image_id** – ID of the image
- **concepts** – a list of concepts
- **not_concepts** – a list of concepts
- **crop** – crop information, with four corner coordinates
- **metadata** – meta data with a dictionary
- **geo** – geo info with a dictionary
- **allow_duplicate_url** – True of False, the flag to allow duplicate url to be imported

Returns the image object just got created and uploaded

Examples::

```
>>> app.inputs.create_image_bytes(img_bytes="raw image bytes...")
```

create_image_from_filename (*filename*, *image_id=None*, *concepts=None*, *not_concepts=None*, *crop=None*, *metadata=None*, *geo=None*, *allow_duplicate_url=False*)

create an image by local filename

Parameters

- **filename** – local filename
- **image_id** – ID of the image
- **concepts** – a list of concepts
- **not_concepts** – a list of concepts
- **crop** – crop information, with four corner coordinates
- **metadata** – meta data with a dictionary
- **geo** – geo info with a dictionary
- **allow_duplicate_url** – True of False, the flag to allow duplicate url to be imported

Returns the image object just got created and uploaded

Examples::

```
>>> app.inputs.create_image_filename(filename="a.jpeg")
```

create_image_from_url (*url*, *image_id=None*, *concepts=None*, *not_concepts=None*, *crop=None*, *metadata=None*, *geo=None*, *allow_duplicate_url=False*)

create an image from Image url

Parameters

- **url** – image url
- **image_id** – ID of the image
- **concepts** – a list of concepts
- **not_concepts** – a list of concepts
- **crop** – crop information, with four corner coordinates
- **metadata** – meta data with a dictionary
- **geo** – geo info with a dictionary
- **allow_duplicate_url** – True of False, the flag to allow duplicate url to be imported

Returns the image object just got created and uploaded

Examples::

```
>>> app.inputs.create_image_from_url(url='https://samples.clarifai.com/
↪metro-north.jpg')
>>>
>>> # create image with geo point
>>> app.inputs.create_image_from_url(url='https://samples.clarifai.com/
↪metro-north.jpg',       >>>    geo=Geo(geo_point=GeoPoint(22.22, 44.44))
```

delete(*input_id*)
delete an input with input ID

Parameters **input_id** – the unique input ID

Returns ApiStatus object

Examples

```
>>> ret = app.inputs.delete('id1')
>>> print ret.code
```

delete_all()
delete all inputs from the application

delete_concepts(*input_id*, *concepts*)
delete concepts from an input/image

Parameters

- **input_id** – unique ID of the input
- **concepts** – a list of concept name

Returns an Image object

get(*input_id*)
get an Input object by input ID

Parameters **input_id** – the unique identifier of the input

Returns an Image/Input object

Examples

```
>>> image = app.inputs.get('id1')
>>> print image.input_id
```

get_all()
get all inputs in a generator

Parameters **Void** –

Returns a generator that yields Input object

Examples

```
>>> for image in app.inputs.get_all():
>>>   print image.input_id
```

get_by_page(*page=1*, *per_page=20*)
get input with pagination

Parameters

- **page** – page number
- **per_page** – number of inputs to retrieve per page

Returns a list of Input object

Examples

```
>>> for image in app.inputs.get_by_page(2, 10):
>>>   print image.input_id
```

get_outputs(*input_id*)
get output predictions for a particular input

Parameters **input_id** – the unique identifier of the input

Returns the input with the output predictions

merge_concepts(*input_id*, *concepts*, *not_concepts*, *overwrite=False*)
merge concepts for one input

Parameters

- **input_id** – the unique ID of the input
- **concepts** – the list of concepts
- **not_concepts** – the list of negative concepts

Returns an Input object

Examples

```
>>> app.inputs.merge_concepts('id', ['cat', 'kitty'], ['dog'])
```

merge_metadata(*input_id*, *metadata*)
merge metadata for the image

This is to merge/update the metadata of the given image

Parameters

- **input_id** – the unique ID of the input
- **metadata** – the metadata dictionary

Examples

```
>>> # merge the metadata
>>> # metadata will be merged along with the existing key/value
>>> app.inputs.merge_metadata('id', {'key1':'value1', 'key2':'value2'})
```

merge_outputs_concepts(*input_id*, *concept_ids*)

merge new concepts into the outputs predictions the concept ids must be present in your app

Parameters

- **input_id** – the unique identifier of the input

- **concept_ids** – the list of concept ids that are present in your app

Returns the patched input in JSON object

remove_outputs_concepts(*input_id*, *concept_ids*)

remove concepts from the outputs predictions the concept ids must be present in your app

Parameters

- **input_id** – the unique identifier of the input
- **concept_ids** – the list of concept ids that are present in your app

Returns the patched input in JSON object

search(*qb*, *page=1*, *per_page=20*)
search with a clarifai image query builder

WARNING: this is the advanced search function. You will need to build a query builder in order to use this.

There are a few simple search functions: search_by_annotated_concepts() search_by_predicted_concepts() search_by_image() search_by_metadata()

Parameters **qb** – clarifai query builder

Returns a list of Input/Image object

search_by_annotated_concepts(*concept=None*, *concepts=None*, *value=True*, *values=None*, *concept_id=None*, *concept_ids=None*, *page=1*, *per_page=20*)
search over the user annotated concepts

Parameters

- **concept** – concept name to search
- **concepts** – a list of concept name to search
- **concept_id** – concept id to search
- **concept_ids** – a list of concept id to search
- **value** – whether the concept should exist or NOT
- **values** – the list of values corresponding to the concepts
- **page** – page number
- **per_page** – number of images to return per page

Returns a list of Image object

Examples

```
>>> app.inputs.search_by_annotated_concepts(concept='cat')
```

search_by_geo(*geo_point=None*, *geo_limit=None*, *geo_box=None*, *page=1*, *per_page=20*)
search by geo point and geo limit

Parameters

- **geo_point** – A GeoPoint object, which represents the (longitude, latitude) of a location
- **geo_limit** – A GeoLimit object, which represents a range to a GeoPoint
- **geo_box** – A GeoBox object, wihch represents a box area

Returns a list of Image object

Examples

```
>>> app.inputs.search_by_geo(GeoPoint(30, 40), GeoLimit("mile", 10))
```

search_by_image(*image_id=None*, *image=None*, *url=None*, *imgbytes=None*, *base64bytes=None*, *fileobj=None*, *filename=None*, *crop=None*, *page=1*, *per_page=20*)

search for visually similar images

By passing image_id, raw image bytes, base64 encoded bytes, image file io stream, image filename, or Clarifai Image object, you can use the visual search power of the Clarifai API.

Also you can specify crop of the image to search over

Parameters

- **image_id** – unique ID of the image for search
- **image** – Image object for search
- **imgbytes** – raw image bytes for search
- **base64bytes** – base63 encoded image bytes
- **fileobj** – file io stream, like open(file)
- **filename** – filename on local filesystem
- **crop** – crop of the image
- **page** – page number
- **per_page** – number of images returned per page

Returns a list of Image object

Examples

```
>>> # search by image url
>>> app.inputs.search_by_image(url='http://blabla')
>>> # search by local filename
>>> app.inputs.search_by_image(filename='bla')
>>> # search by raw image bytes
>>> app.inputs.search_by_image(imgbytes='data')
>>> # search by base64 encoded image bytes
>>> app.inputs.search_by_image(base64bytes='data')
>>> # search by file stream io
>>> app.inputs.search_by_image(fileobj=open('file'))
```

search_by_metadata(*metadata*, *page=1*, *per_page=20*)

search by other meta data of the image rather than concept

Parameters

- **metadata** – is a dictionary for meta data search. The dictionary could be a simple one with only one key and value, Or a nested dictionary with multi levels.
- **page** – page number
- **per_page** – the number of images to return per page

Returns a list of Image object

Examples

```
>>> app.inputs.search_by_metadata(metadata={'name':'bla'})
>>> app.inputs.search_by_metadata(metadata={'my_class1': { 'name' : 'bla' }})
```

search_by_original_url(*url*, *page=1*, *per_page=20*)

search by original url of the imported images

Parameters

- **url** – url of the image
- **page** – page number
- **per_page** – the number of images to return per page

Returns a list of Image object

Examples

```
>>> app.inputs.search_by_original_url(url='http://bla')
```

search_by_predicted_concepts(*concept=None*, *concepts=None*, *value=True*, *values=None*, *concept_id=None*, *concept_ids=None*, *page=1*, *per_page=20*, *lang=None*)

search over the predicted concepts

Parameters

- **concept** – concept name to search
- **concepts** – a list of concept name to search
- **concept_id** – concept id to search
- **concept_ids** – a list of concept id to search
- **value** – whether the concept should exist or NOT
- **values** – the list of values corresponding to the concepts
- **page** – page number
- **per_page** – number of images to return per page
- **lang** – language to search over for translated concepts

Returns a list of Image object

Examples

```
>>> app.inputs.search_by_predicted_concepts(concept='cat')
>>> # search over simplified Chinese label
>>> app.inputs.search_by_predicted_concepts(concept=u'', lang='zh')
```

update(*image*, *action='merge'*)
update the input update the information of an input/image

Parameters

- **image** – an Image() object that has concepts, metadata, etc.
- **method** – one of ['merge', 'overwrite'] 'merge' is to merge the info into the exising info, for either concept or metadata 'overwrite' is to overwrite the metadata, concepts with the existing ones

Returns an Image object

Examples

```
>>> new_img = Image(image_id="abc", concepts=['c1', 'c2'], not_concepts=['c3
↪'], metadata={'key':'val'})
>>> app.inputs.update(new_img, action='overwrite')
```

Models

class clarifai.rest.client.**Models**(*api*)

clear_model_cache()
clear model_name -> model_id cache

WARNING: This is an internal function, user should not call this

We cache model_name to model_id mapping for API efficiency At the first time you call a models.get() by name, the name to ID mapping is saved so next time there is no query. Then user does not have to query the model ID every time when they want to work on it.

Returns There is no return result for this call

create(*model_id*, *model_name=None*, *concepts=None*, *concepts_mutually_exclusive=False*, *closed_environment=False*, *hyper_parameters=None*)
create a new model

Parameters

- **model_id** – ID of the model
- **model_name** – optional name of the model
- **concepts** – optional concepts to associated with this model
- **concepts_mutually_exclusive** – True or False, whether concepts are mutually exclusive
- **closed_environment** – True or False, whether use negatives for prediction
- **hyper_parameters** – hyper parameters for the model, with a json object

Returns Model object

Examples

```
>>> # create a model with no concepts
>>> app.models.create('my_model1')
>>> # create a model with a few concepts
>>> app.models.create('my_model2', concepts=['bird', 'fish'])
>>> # create a model with closed environment
>>> app.models.create('my_model3', closed_environment=True)
```

delete(*model_id*, *version_id=None*)
delete the model, or a specific version of the model

Without model version id specified, it is to delete a model. Then all the versions associated with this model will be deleted as well.

With model version id specified, it is to delete a particular model version from the model

Parameters

- **model_id** – the unique ID of the model
- **version_id** – the unique ID of the model version

Returns the raw JSON response from the server

Examples

```
>>> # delete a model
>>> app.models.delete('model_id1')
>>> # delete a model version
>>> app.models.delete('model_id1', version_id='version1')
```

delete_all()
delete all models and the versions associated with each one

After this operation, you will have no model in the application

Returns the raw JSON response from the server

Examples

```
>>> app.models.delete_all()
```

get(*model_id*, *model_type=None*)
get a model, by ID or name

Parameters

- **model_id** – unique identifier of a model
- **model_type** – type of the model

Returns the Model object

Examples

```
>>> # get general-v1.3 model
>>> app.models.get('general-v1.3')
```

get_all (*public_only=False*, *private_only=False*)
get all models in the application

Parameters

- **public_only** – only yield public models
- **private_only** – only yield private models that tie to your own account

Returns a generator that yields Model object

Examples

```
>>> for model in app.models.get_all():
>>>   print model.model_name
```

get_by_page (*public_only=False*, *private_only=False*, *page=1*, *per_page=20*)
get paginated models from the application

When the number of models get high, you may want to get the paginated results from all the models

Parameters

- **public_only** – only yield public models
- **private_only** – only yield private models that tie to your own account
- **page** – page number
- **per_page** – number of models returned in one page

Returns a list of Model objects

Examples

```
>>> models = app.models.get_by_page(2, 20)
```

init_model_cache ()
initialize the model cache for the public models

This will go through all public models and cache them

search (*model_name*, *model_type=None*)
search model by name and type

search the model by name, default is to search concept model only. All the custom model trained are concept model.

Parameters

- **model_name** – name of the model. name is not unique.
- **model_type** – default to None, equivalent to wildcards search

Returns a list of Model objects or None

Examples

```
>>> # search for general-v1.3 models
>>> app.models.search('general-v1.3')
>>>
>>> # search for color model
>>> app.models.search('color', model_type='color')
>>>
>>> # search for face model
>>> app.models.search('face-v1.3', model_type='facedetect')
```

Model

class clarifai.rest.client.**Model**(*api*, *item=None*)

add_concepts(*concept_ids*)
merge concepts in a model

This is just an alias of *merge_concepts*, for easier understanding of adding new concepts to the model without overwritting them

Parameters **concept_ids** – a list of concept id

Returns the Model object

Examples

```
>>> model = self.app.models.get('model_id')
>>> model.add_concepts(['cat', 'dog'])
```

delete_concepts(*concept_ids*)
delete concepts from a model

Parameters **concept_ids** – a list of concept id

Returns the Model object

Examples

```
>>> model = self.app.models.get('model_id')
>>> model.delete_concepts(['cat', 'dog'])
```

delete_version(*version_id*)
delete model version by version_id

Parameters **version_id** – version id of the model version

Returns the JSON response

Examples

```
>>> model = self.app.models.get('model_id')
>>> model.delete_version('model_version_id')
```

get_concept_ids()
get concepts IDs associated with the model

Parameters **Void** –

Returns a list of concept IDs

Examples

```
>>> ids = model.get_concept_ids()
```

get_info(*verbose=False*)
get model info, with or without concepts info

Parameters **verbose** – default is False. True will yield output_info, with concepts of the model

Returns raw json of the response

Examples

```
>>> # with basic model info
>>> model.get_info()
>>> # model info with concepts
>>> model.get_info(verbose=True)
```

get_inputs(*version_id=None*, *page=1*, *per_page=20*)

get all the inputs from the model or a specific model version Without specifying model version id, this will yield the inputs

Parameters

- **version_id** – model version id
- **page** – page number
- **per_page** – number of inputs to return for each page

Returns A list of Input objects

get_version(*version_id*)
get model version info for a particular version

Parameters **version_id** – version id of the model version

Returns the JSON response

Examples

```
>>> model = self.app.models.get('model_id')
>>> model.get_version('model_version_id')
```

list_versions()
list all model versions

Parameters **void** –

Returns the JSON response

Examples

```
>>> model = self.app.models.get('model_id')
>>> model.list_versions()
```

merge_concepts(*concept_ids*, *overwrite=False*)
merge concepts in a model

If the concept does not exist in the model, it will be appended, otherwise, the original one will be kept

Parameters

- **concept_ids** – a list of concept id
- **overwrite** – True of False. If True, the concepts will be overwritten

Returns the Model object

predict(*inputs*, *model_output_info=None*)
predict with multiple images

Parameters **inputs** – a list of Image object

Returns the prediction of the model in JSON format

predict_by_base64(*base64_bytes*, *lang=None*, *is_video=False*)
predict a model with base64 encoded image bytes

Parameters

- **base64_bytes** – base64 encoded image bytes
- **lang** – language to predict, if the translation is available
- **is_video** – whether this is a video

Returns the prediction of the model in JSON format

predict_by_bytes(*raw_bytes*, *lang=None*, *is_video=False*)
predict a model with image raw bytes

Parameters

- **raw_bytes** – raw bytes of an image
- **lang** – language to predict, if the translation is available
- **is_video** – whether this is a video

Returns the prediction of the model in JSON format

predict_by_filename (*filename*, *lang=None*, *is_video=False*)
predict a model with a local filename

Parameters

- **filename** – filename on local filesystem
- **lang** – language to predict, if the translation is available
- **is_video** – whether this is a video

Returns the prediction of the model in JSON format

predict_by_url (*url*, *lang=None*, *is_video=False*)
predict a model with url

Parameters

- **url** – url of an image
- **lang** – language to predict, if the translation is available
- **is_video** – whether this is a video

Returns the prediction of the model in JSON format

train (*sync=True*, *timeout=60*)
train a model

train the model in synchronous or asynchronous mode

Parameters **sync** – indicating synchronous or asynchronous, default is True

Returns the Model object

update (*action='merge'*, *model_name=None*, *concepts_mutually_exclusive=None*, *closed_environment=None*, *concept_ids=None*)
update the model attributes

This is to update the model attributes. The name of the model, and list of concepts could be changed. Also the training attributes concepts_mutually_exclusive and closed_environment could be changed. Note this is a overwriting change. For a valid call, at least one or more attributes should be specified. Otherwise the call will be just skipped without error.

Parameters

- **action** – the way to patch the model: ['merge', 'remove', 'overwrite']
- **model_name** – name of the model
- **concepts_mutually_exclusive** – whether it's multually exclusive model
- **closed_environment** – whether it's closed environment training
- **concept_ids** – a list of concept ids

Returns the Model object

Examples

```
>>> model = self.app.models.get('model_id')
>>> model.update(model_name="new_model_name")
>>> model.update(concepts_mutually_exclusive=False)
>>> model.update(closed_environment=True)
```

```
>>> model.update(concept_ids=["bird", "hurd"])
>>> model.update(concepts_mutually_exclusive=True, concept_ids=["bird", "hurd
↪"])
```

Search Syntax

class clarifai.rest.client.**SearchTerm**
Clarifai search term interface the base class for InputSearchTerm and OutputSearchTerm

It is used to build SearchQueryBuilder

class clarifai.rest.client.**InputSearchTerm**(*url=None*, *input_id=None*, *concept=None*, *concept_id=None*, *value=True*, *metadata=None*, *geo=None*)
Clarifai Image Search Input search term For input search, you can specify search term for url string match, input_id string match, concept string match, and concept_id string match value indicates whether the concept search is a NOT search

Examples

```
>>> # search for url, string match
>>> InputSearchTerm(url='http://blabla')
>>> # search for input ID, string match
>>> InputSearchTerm(input_id='site1_bla')
>>> # search for annotated concept
>>> InputSearchTerm(concept='tag1')
>>> # search for not the annotated concept
>>> InputSearchTerm(concept='tag1', value=False)
>>> # search for metadata
>>> InputSearchTerm(metadata={'key':'value'})
>>> # search for geo
>>> InputSearchTerm(geo=Geo(geo_point=GeoPoint(-40, 30), geo_limit=GeoLimit(
↪'withinMiles', 10)))
```

class clarifai.rest.client.**OutputSearchTerm**(*url=None*, *base64=None*, *input_id=None*, *concept=None*, *concept_id=None*, *value=True*, *crop=None*)
Clarifai Image Search Output search term For output search, you can specify search term for url, base64, and input_id for visual search, or specify concept and concept_id for string match value indicates whether the concept search is a NOT search

Examples

```
>>> # search for visual similarity from url
>>> OutputSearchTerm(url='http://blabla')
>>> # search for visual similarity from base64 encoded image
>>> OutputSearchTerm(base64='sdfds')
>>> # search for visual similarity from input id
>>> OutputSearchTerm(input_id='site1_bla')
>>> # search for predicted concept
>>> OutputSearchTerm(concept='tag1')
>>> # search for not the predicted concept
>>> OutputSearchTerm(concept='tag1', value=False)
```

Authentication

class clarifai.rest.client.**Auth**(*api*)
Clarifai Authentication

This class is initialized as an attribute of the clarifai application object with app.auth

get_token()
get token string

Returns The token as a string

Exceptions

class clarifai.rest.client.**ApiError**(*resource*, *params*, *method*, *response*)
API Server error

class clarifai.rest.client.**ApiClientError**
API Client Error

class clarifai.rest.client.**UserError**
User Error

Index