Python

Release

Sep 21, 2017

Contents

class `flowthings.`**`Token`**(*account*, *token*)

Token objects may be passed to an *`API`*. This will be used to sign requests to the platform.

```
>>> creds = Token('<account>', '<token>')
```

static **`from_bluemix`**(*default=None*, *env_var='VCAP_SERVICES'*)

Loads a *`Token`* object from an IBM Bluemix environment. If a `default` *`Token`* is provided, it will be returned in case of a failure, otherwise a *`FlowThingsError`* will be raised.

class `flowthings.`**`API`**(*creds*, *async_lib=DEFAULT*, *secure=DEFAULT*, *params=DEFAULT*)

Creates a new API instance for interacting with the platform.

```
>>> api = API(creds)
```

Options labelled as `DEFAULT` will use the options set in the *`defaults`* configuration object.

An *`API`* is comprised of services for querying the different domains on the platform:

- `flow`
- `drop`
- `track`
- `group`
- `identity`
- `api_task`
- `mqtt_task`
- `rss_task`
- `token`
- `share`
- `device`
- `statistics`
- `websocket`

For documentation on these services, read *Service Methods*, *Authentication*, *Statistics*, *Aggregation*, and *Web-Sockets*.

`async`([*pool*])

Returns an API wrapper for making asynchronous requests using either `eventlet` or `gevent`. Requests made using an *`async()`* API will return green threads.

For more documentation, read *Asynchronous and Parallel Requests*.

`lazy`([*pool*])

Returns an API wrapper for making implicitly parallel requests using either `eventlet` or `gevent`. Requests made using a *`lazy()`* API will return thunks that wait on their respective green thread when accessed.

For more documentation, read *Asynchronous and Parallel Requests*.

`request`(*method*, *path*, *data=None*, *params=None*)

Parameters

- **`method`** (*`str`*) – HTTP method

- **path** (*str*) – Request path
- **data** (*dict*) – Request data
- **params** (*dict*) – Request query parameters

Makes an arbitrary platform request.

creds

Get or set the API's *Token*.

flowthings.**defaults**

Configuration object for globally setting default options for *API* instances.

defaults.**async_lib**

Defaults to `None`. Supports `eventlet` and `gevent`.

```
import eventlet

flowthings.defaults.async_lib = eventlet
```

defaults.**secure**

Defaults to `True`. When set to `False`, requests will be made over `http://` rather than `https://`.

defaults.**params**

The default set of query string parameters sent with all requests. Defaults to `{}`.

CHAPTER 1

Service Methods

All `API` service requests return plain dictionaries of the request body. They may throw *exceptions* in case of an error.

`service.`**`read`**`(`*id*, ***params*`)`

> **Parameters** **`id`** (*`str`*) – The resource id

```
>>> api.flow.read('<flow_id>')
```

`service.`**`read_or_else`**`(`*id*, *default=None*, ***params*`)`

> **Parameters**
> - **`id`** (*`str`*) – The resource id
> - **`default`** (*`any`*) – Default value when the resource is not found

```
>>> api.flow.read_or_else('<flow_id>', None)
```

`service.`**`read_many`**`(`*ids*, ***params*`)`

> **Parameters** **`ids`** (*`list`*) – List of resource ids

```
>>> api.flow.read_many(['<flow_id_1>', '<flow_id_2'])
```

`service.`**`find_many`**`(`**filters*, ***params*`)`

> **Parameters** **`filters`** (*`Filter`*) – Request filters

```
>>> api.flow.find_many(mem.displayName == 'Foo')
```

`service.`**`find`**`(`..., ***params*`)`

An overloaded method which may call one of *`read()`*, *`read_many()`*, or *`find_many()`* depending upon the type of the first argument.

```
>>> api.flow.find('<flow_id>')
>>> api.flow.find(['<flow_id_1>', '<flow_id_2'])
>>> api.flow.find(mem.displayName == 'Foo')
```

`service.`**`create`**(*model*, ***params*)

Parameters **`model`** (*`dict`*) – Initial data for a new resource

```
>>> api.flow.create({'path': '/path/to/flow'})
```

`service.`**`update`**(*model*, ***params*)

Parameters **`model`** (dict or *`M`*) – Updated model

Requests are made based on the model's `'id'` key.

```
>>> api.flow.update({'id': '<flow_id>', 'displayName': 'Foo'})
>>> api.flow.update(M(model, displayName='Foo'))
```

`service.`**`update_many`**(*models*, ***params*)

Parameters **`models`** (*`list`*) – List of updated models

`service.`**`save`**(..., ***params*)

An overloaded method which may call one of *`create()`*, *`update()`*, or *`update_many()`* depending upon the type of the first argument. *`create()`* or *`update()`* are called based on the presence of an `'id'` key.

`service.`**`delete`**(*id*, *data=None*, ***params*)

Parameters

- **`id`** (*`str`*) – The resource to delete
- **`data`** (*`any`*) – Request data

```
>>> api.flow.delete('<flow_id>')
```

Note: The `drop` service is slightly different in that it must first be parameterized by the Flow id.

```
>>> api.drop('<flow_id>').find(limit=10)
```

CHAPTER 2

Request Parameters

Service methods take additional keyword arguments that act as query parameters on the requests. These are not fixed in any way, so please refer to the platform documentation for the options.

Note: When a request is made with the `refs` parameter set to `True`, the return type becomes a tuple rather than a single dictionary:

```
>>> resp, refs = api.flow.find('<flow_id>', refs=True)
```

CHAPTER 3

Request Filters

Service find methods understand a query DSL that lets you express filters using Python operations instead of manually splicing strings together.

```
>>> api.flow.find(mem.displayName == 'foo', mem.path.re('^/foo', 'i'))
```

class `flowthings.`**`mem`**

mem represents members of the objects you are querying. You can use use properties or key indexing to represent a member.:

```
>>> api.drop(<flow_id>).find(mem.elems.foo > 12)
```

The supported operators are ==, <, <=, >, and >= along with the following methods, mirroring the platform:

`re`(*pattern*[, *flags*])

`IN`(**items*)

`CONTAINS`(**items*)

`WITHIN`(*distance*, *unit*[, *coords=(lat*, *lon)*[, *zip=zipcode*]])

Additional platform filter operations are supported:

`flowthings.`**`EXISTS`**(*member*)

`flowthings.`**`HAS`**(*elem_type*)

`flowthings.`**`MATCHES`**(*pattern*[, *flags*])

`flowthings.`**`NOT`**(*filter*)

`flowthings.`**`AGE`**

Age comparisons can be made using normal python operators with `AGE`.:

```
>>> api.flow.find(AGE > time_millis)
```

Boolean operations are supported on filters using `AND` and `OR`.:

```
>>> api.flow.find((mem.displayName == 'foo').OR(mem.displayName == 'bar'))
```

CHAPTER 4

Authentication

If you create your `API` using a master token, you can create and manage tokens and shares.

`api.token.`**`create`**`(`*model*, ***params*`)`

`api.share.`**`create`**`(`*model*, ***params*`)`

Both tokens and shares support `find` and `delete` methods like other services. They are, however, immutable and do not support updates.

CHAPTER 5

Statistics

`api.statistics.`**`flow_drop_added`**(*flow_id*, *year=None*, *month=None*, *day=None*, *level=None*)

`api.statistics.`**`flow_tracked`**(*flow_id*, *year=None*, *month=None*, *day=None*, *level=None*)

`api.statistics.`**`track_hit`**(*track_id*, *year=None*, *month=None*, *day=None*, *level=None*)

`api.statistics.`**`track_pass`**(*track_id*, *year=None*, *month=None*, *day=None*, *level=None*)

`api.statistics.`**`api_call_by_identity`**(*identity_id*, *year=None*, *month=None*, *day=None*, *level=None*)

`api.statistics.`**`drop_created_by`**(*identity_id*, *year=None*, *month=None*, *day=None*, *level=None*)

CHAPTER 6

Aggregation

`api.`**`drop`** (*flow_id).aggregate(outputs, group_by=None, filter=None, rules=None, sorts=None*)

Both `filter` and `rules` support *Request Filters*.:

```
>>> api.drop(flow_id).aggregate(['$avg:test'], rules={'test': mem.foo > 42})
```

CHAPTER 7

Exceptions

class `flowthings.`**`FlowThingsError`**

class `flowthings.`**`FlowThingsException`**

> **`errors`**
> List of errors returned from the platform
>
> **`creds`**
> Request credentials
>
> **`method`**
> Request HTTP method
>
> **`path`**
> Request path

class `flowthings.`**`FlowThingsBadRequest`**

class `flowthings.`**`FlowThingsForbidden`**

class `flowthings.`**`FlowThingsNotFound`**

class `flowthings.`**`FlowThingsServerError`**

CHAPTER 8

Modifications

`Service update methods` can also take an instance of a modification helper called *`M`*. It lets you gradually make updates to a model and then extract the diff and model with the changes applied.

When passed directly to an update method, only the changes will be sent to the server instead of the entire model.

class `flowthings.`**`M`**`(`*model*, ***changes*`)`

> **`modify`**`(`*key*, *val*`)`
>
> **`done`**`()`
>
> Returns a tuple of `(new_model, diff)`.

CHAPTER 9

Asynchronous and Parallel Requests

Two workflows are supported for making asynchronous and parallel requests.

The *API.async()* workflow is an imperative API where requests are queued internally. Once you've made all the requests you need, you can invoke the `results()` method to wait. This can be useful when making large batches of similar requests:

```
paths = [...]
async_api = api.async()

for path in paths
    async_api.flow.find(mem.path == path)

for flows in async_api.results():
    # Do something with the flows
    pass
```

If some of your requests might fail, and you want to know which ones, you may set the `with_exceptions` keyword argument:

```
flows = [...]
async_api = api.async()

for flow in flows:
    async_api.drop(flow['id']).find(limit=10)

for e, drops in async_api.results(with_exceptions=True):
    if e:
        # Do something if there was an error
        pass
    else:
        # Do something with the drops
        pass
```

The *API.lazy()* worklow is useful when building complex compositions of dependent requests which can benefit from implicit parallelization. All requests are executed in parallel, but wait when you try to read the data. This works by requests returning a `GreenThunk`, which is a `MutableMapping` around a green thread. This object acts just like a regular dictionary or list, but waits on the green thread before performing any look-ups or mutations.

```
lazy_api = api.lazy()
flow_a = lazy_api.flow.find(mem.path == '/path/to/flow_a')
flow_b = lazy_api.flow.find(mem.path == '/path/to/flow_b')
drops  = lazy_api.drop(flow_a[0]['id']).find(limit=10)
```

In this example, the two requests for Flows are performed in parallel, while the requests for drops waits for the `flow_a` request to complete first.

You can retrieve the pure data of a `GreenThunk` by invoking its `unwrap()` method.

Note: It is assumed the user has done the necessary green thread monkey-patching for their chosen library before importing the `flowthings` package.

CHAPTER 10

WebSockets

WebSockets are supported using the `websocket-client` package. Here is a short example:

```
def on_open(ws):
    ws.subscribe('<flow_id>')

def on_message(ws, resource, data):
    print 'Got message:', resource, data

def on_close(ws):
    print 'Closed'

def on_error(ws, e):
    print 'Error:', e

ws = api.websocket.connect(on_open=on_open,
                           on_message=on_message,
                           on_close=on_close,
                           on_error=on_error)
ws.run()
```

CHAPTER 11

Examples

```python
from flowthings import API, Token, mem

creds = Token('<account_name>', '<token_string>')
api = API(creds)

# Get a Flow by id
api.flow.find('<flow_id>')

# Get a Flow by path
api.flow.find(mem.path == '<flow_path>')

# Get 10 recent Flows, with references
flows, refs = api.flow.find(limit=10, refs=True)

# Create a flow
api.flow.create({ 'path': '<flow_path' })

# Delete a flow
api.flow.delete('<flow_id>')

# Get drops in a flow
api.drop('<flow_id>').find()

# Filter drops in a flow
api.drop('<flow_id>').find(mem.elems.foo == 'value')
```

Python Module Index

Index