mocurly Documentation

Release 0.1.3

Captricity

September 30, 2014

Contents

Author Captricity

Version 0.1.3

Date 2014/09/03

Homepage Mocurly Homepage

Download PyPI

License MIT License

Change log *change-log*

Issue tracker Github issues

Mocurly is a library that mocks the Recurly Python client so that you can easily write tests for applications that use the Recurly Python client.

Mocurly acts as a mock backend for the Recurly client, allowing you to use the Recurly Python client AS IS. This means that all your code that uses the Recurly Python client and targets Recurly objects will all work as you would expect. Best of all: you can use the Recurly Python client to setup the test environment!

For example, suppose you had a simple function in your app that lists all the users in Recurly, and counts them:

```
import recurly
recurly.API_KEY = 'foo'
recurly.SUBDOMAIN = 'bar'

def count_recurly_accounts():
    return len(recurly.Account.all())
```

With Mocurly, you can test the above code like so:

```
import recurly
recurly.API_KEY = 'foo'
recurly.SUBDOMAIN = 'bar'
from mocurly import mocurly
from count_module import count_recurly_accounts

@mocurly
def test_count_recurly_accounts():
    for i in range(10):
        recurly.Account(account_code=str(i)).save()
    assert count_recurly_accounts() == 10
```

Within the decorator context, all calls to Recurly are captured by Mocurly, which keeps the state in memory for the duration of the context. No actual web calls are made, allowing you to test your Recurly code without worrying about existing context or internet connections.

CHAPTER 1

Usage

1.1 API Documentation

class mocurly.**mocurly**(*func=None*)

Main class that provides the mocked context.

This can be used as a decorator, as a context manager, or manually. In all three cases, the guarded context will route all recurly requests to the mocked callback functions defined in endpoints.py.

register_transaction_failure(*account_code*, *error_code*)

Register a transaction failure for the given account.

This will setup mocurly such that all transactions made by the account with the given *account_code* will fail with the selected *error_code*.

start()

Starts the mocked context by enabling HTTPretty to route requests to the defined endpoints.

start_timeout(*timeout_filter=None*)

Notifies mocurly to start simulating time outs within the current context.

You can pass in a filter function which will be used to decide what requests to timeout. The function will get one parameter, *request*, which is an instance of the HTTPrettyRequest class, and should return a boolean which when True, the request will time out.

The following attributes are available on the request object: *headers* -> a mimetype object that can be cast into a dictionary, contains all the request headers.

method -> the HTTP method used in this request.

path -> the full path to the requested URI.

querystring -> a dictionary containing lists with the attributes.

body -> the raw contents of the request body.

***parsed_body* -> a dictionary containing parsed request body or** *None* if *HTTPrettyRequest* doesn't know how to parse it.

start_timeout_successful_post(*timeout_filter=None*)

Notifies mocurly to make timeouts on POST requests, but only after it has caused state changes.

Like *start_timeout*, you can pass in a filter function used to decide which requests to cause the timeout on.

stop()

Stops the mocked context, restoring the routes back to what they were

stop_timeout()
Notifies mocurly to stop simulating time outs within the current context.

stop_timeout_successful_post()
Notifies mocurly to stop simulating successful POST time outs within the current context.

1.2 Basic Usage

Mocurly is designed to be used as a wrapper around blocks of code that needs the mocked Recurly context. Within the context, all calls made using the Recurly Python client will talk to the mocked in-memory service instead of the real Recurly.

In the following example, the call to the `save()` method of the `recurly.Account` class will create an instance of the account object in Mocurly's in-memory database, but not in your Recurly account:

```
>>> import recurly
>>> recurly.API_KEY = 'foo'
>>> recurly.SUBDOMAIN = 'bar'
>>> from mocurly import mocurly
>>> with mocurly():
>>>     recurly.Account(account_code='foo').save()
```

Note that you still have to set the *API_KEY* and *SUBDOMAIN* on the Recurly instance, since the Recurly client itself has assertions to make sure they are set. However, the values you use do not matter. They also have to be set outside the Mocurly context, as in the example.

Mocurly can be used as a decorator, context manager, or manually. In all 3 cases, the Mocurly context is reset at the start of the invocation.

1.2.1 Mocurly as decorator

```
@mocurly
def test_count_recurly_accounts():
    for i in range(10):
        recurly.Account(account_code=str(i)).save()
    assert count_recurly_accounts() == 10
```

1.2.2 Mocurly as context manager

```
def test_count_recurly_accounts():
    with mocurly():
        for i in range(10):
            recurly.Account(account_code=str(i)).save()
        assert count_recurly_accounts() == 10
```

1.2.3 Mocurly used manually

```
def test_count_recurly_accounts():
    mocurly_ = mocurly()
    mocurly_.start()

    for i in range(10):
```

```
        recurly.Account(account_code=str(i)).save()
    assert count_recurly_accounts() == 10

    mocurly_.stop()
```

1.3 Advanced Usage

1.3.1 Error handling

Mocurly supports simulating certain error scenarios that commonly occur through the use of Recurly. Currently, as of version 0.1.3, Mocurly supports the following three scenarios:

- Request timeout
- Request timeout with a successful POST
- Declined transactions

Note that at this time, error handling is only supported in manual usage.

Request timeout

Sometimes API calls may timeout. This may be due to faulty internet connections, or a server outage at recurly.com. While this is an unlikely case, we want to be prepared for those scenarios. That is why Mocurly supports simulating a request that times out.

To trigger timeouts for requests, you can use the `start_timeout()` method on your *mocurly* instance, assuming the context has been started. When called, this will trigger Mocurly to always simulate a timeout by raising an `ssl.SSLError` exception for all requests.

You can stop this behavior at any point by calling the `stop_timeout()` method.

In the following code snippet, the first `save()` call will raise an `ssl.SSLError` because it is wrapped in the timeout context, but the second will succeed:

```
>>> mocurly_ = mocurly()
>>> mocurly_.start()
>>> mocurly_.start_timeout()
>>> # Will fail
>>> recurly.Account(account_code='foo').save()
>>> mocurly_.stop_timeout()
>>> # Will succeed
>>> recurly.Account(account_code='foo').save()
>>> mocurly_.stop()
```

You can also pass an optional filter function to `start_timeout()` to control what requests get timed out. The filter function will receive a *request* object, which contains several attributes you can use to filter the call on. Refer to the API reference for more details on what attributes the request object contains.

For example, the following snippet will only trigger timeouts on GET requests:

```
>>> mocurly_ = mocurly()
>>> mocurly_.start()
>>> mocurly_.start_timeout(lambda request: request.method == 'GET')
>>> # Will succeed
>>> recurly.Account(account_code='foo').save()
>>> # Will fail
```

```
>>> recurly.Account.get('foo')
>>> mocurly_.stop_timeout()
>>> mocurly_.stop()
```

Request timeout with a successful POST

As mentioned in the previous section, Mocurly supports simulating requests that timeout. However, sometimes the request times out while receiving a response from the Recurly servers, after the POST request has successfully reached the Recurly server. These cases are especially difficult to deal with, because the Python client raises an exception, even though the request went through and created the object on Recurly's side, which could have charged the user's credit card. Mocurly supports this scenario as well, using a similar syntax as simulating request timeout. To simulate a request timeout with a successfuly POST, use the `start_timeout_successful_post()` method of the Mocurly instance.

Note that by nature of the simulation, this will only simulate the timeout on POST requests. This means that you can use this in combination with `start_timeout()` to simulate complex timeout scenarios.

In the following example, although the save call fails, the account will still be created in the Mocurly database:

```
>>> mocurly_ = mocurly()
>>> mocurly_.start()
>>> mocurly_.start_timeout_successful_post()
>>> # Will fail, but the account is created successfully
>>> recurly.Account(account_code='foo').save()
>>> recurly.Account.get('foo') # returns an actual account
>>> mocurly_.stop_timeout_successful_post()
>>> mocurly_.stop()
```

Like `start_timeout()`, this can also take in a filter function to simulate the timeout only on certain requests.

Declined transactions

When we deal with real credit cards out in the wild, there will always be a case where a transaction might fail due to a declined credit card. Mocurly supports simulating transaction requests with a declined credit card, both for one time transactions and subscription payments.

To trigger a declined credit card, use the `register_transaction_failure()` method on your *mocurly* instance. `register_transaction_failure()` takes two arguments: the *account_code* of the user who's credit card will be rejected, and the error code to use. These error codes should be pulled from *mocurly.errors*. Refer to the API reference for more info on what error codes are available.

In the following snippet, the transaction made for Joe will be declined by Mocurly, but not for Billy:

```
>>> mocurly_ = mocurly()
>>> mocurly_.start()
>>> mocurly_.register_transaction_failure('joe', mocurly.errors.TRANSACTION_DECLINED)
>>> joe = recurly.Account.get('joe')
>>> recurly.Transaction(amount_in_cents=10, currency='USD', account=joe).save() # will fail
>>> billy = recurly.Account.get('joe')
>>> recurly.Transaction(amount_in_cents=10, currency='USD', account=billy).save() # will succeed
>>> mocurly_.stop()
```

LICENSE

m