Betamax Documentation

Release 0.8.0

Ian Cordasco

May 04, 2017

Narrative Documentation

Betamax is a VCR imitation for requests. This will make mocking out requests much easier. It is tested on Travis CI.

Put in a more humorous way: "Betamax records your HTTP interactions so the NSA does not have to."

CHAPTER 1

Example Use

```
from betamax import Betamax
from requests import Session
from unittest import TestCase

with Betamax.configure() as config:
    config.cassette_library_dir = 'tests/fixtures/cassettes'


class TestGitHubAPI(TestCase):
    def setUp(self):
        self.session = Session()
        self.headers.update(...)

    # Set the cassette in a line other than the context declaration
    def test_user(self):
        with Betamax(self.session) as vcr:
            vcr.use_cassette('user')
            resp = self.session.get('https://api.github.com/user',
                                    auth=('user', 'pass'))
            assert resp.json()['login'] is not None

    # Set the cassette in line with the context declaration
    def test_repo(self):
        with Betamax(self.session).use_cassette('repo'):
            resp = self.session.get(
                'https://api.github.com/repos/sigmavirus24/github3.py'
                )
            assert resp.json()['owner'] != {}
```

CHAPTER 2

What does it even do?

If you are unfamiliar with VCR, you might need a better explanation of what Betamax does.

Betamax intercepts every request you make and attempts to find a matching request that has already been intercepted and recorded. Two things can then happen:

1. If there is a matching request, it will return the response that is associated with it.
2. If there is **not** a matching request and it is allowed to record new responses, it will make the request, record the response and return the response.

Recorded requests and corresponding responses - also known as interactions - are stored in files called cassettes. (An example cassette can be seen in the examples section of the documentation.) The directory you store your cassettes in is called your library, or your cassette library.

CHAPTER 3

VCR Cassette Compatibility

Betamax can use any VCR-recorded cassette as of this point in time. The only caveat is that python-requests returns a URL on each response. VCR does not store that in a cassette now but we will. Any VCR-recorded cassette used to playback a response will unfortunately not have a URL attribute on responses that are returned. This is a minor annoyance but not something that can be fixed.

CHAPTER 4

Contributing

You can check out the project board on waffle.io to see what the status of each issue is.

CHAPTER 5

Contents of Betamax's Documentation

Getting Started

The first step is to make sure Betamax is right for you. Let's start by answering the following questions

- Are you using Requests?

  If you're not using Requests, Betamax is not for you. You should checkout VCRpy.

- Are you using Sessions or are you using the functional API (e.g., `requests.get`)?

  If you're using the functional API, and aren't willing to use Sessions, Betamax is not *yet* for you.

So if you're using Requests and you're using Sessions, you're in the right place.

Betamax officially supports py.test and unittest but it should integrate well with nose as well.

Installation

```
$ pip install betamax
```

Configuration

When starting with Betamax, you need to tell it where to store the cassettes that it creates. There's two ways to do this:

1. If you're using `Betamax` or *`use_cassette`* you can pass the `cassette_library_dir` option. For example,

```
import betamax
import requests

session = requests.Session()
recorder = betamax.Betamax(session, cassette_library_dir='cassettes')
```

```
with recorder.use_cassette('introduction'):
    # ...
```

2. You can do it once, globally, for your test suite.

```
import betamax

with betamax.Betamax.configure() as config:
    config.cassette_library_dir = 'cassettes'
```

Note: If you don't set a cassette directory, Betamax won't save cassettes to disk

There are other configuration options that *can* be provided, but this is the only one that is *required*.

Recording Your First Cassette

Let's make a file named `our_first_recorded_session.py`. Let's add the following to our file:

```
import betamax
import requests

CASSETTE_LIBRARY_DIR = 'examples/cassettes/'


def main():
    session = requests.Session()
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('our-first-recorded-session'):
        session.get('https://httpbin.org/get')


if __name__ == '__main__':
    main()
```

If we then run our script, we'll see that a new file is created in our specified cassette directory. It should look something like:

```
{"http_interactions": [{"request": {"body": {"string": "", "encoding": "utf-8"},
↪"headers": {"Connection": ["keep-alive"], "Accept-Encoding": ["gzip, deflate"],
↪"Accept": ["*/*"], "User-Agent": ["python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0
↪"]}, "method": "GET", "uri": "https://httpbin.org/get"}, "response": {"body": {
↪"string": "{\n  \"args\": {}, \n  \"headers\": {\n    \"Accept\": \"*/*\", \n    \
↪"Accept-Encoding\": \"gzip, deflate\", \n    \"Host\": \"httpbin.org\", \n    \
↪"User-Agent\": \"python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0\"\n  }, \n  \
↪"origin\": \"127.0.0.1\", \n  \"url\": \"https://httpbin.org/get\"\n}\n", "encoding
↪": null}, "headers": {"content-length": ["265"], "server": ["nginx"], "connection":␣
↪["keep-alive"], "access-control-allow-credentials": ["true"], "date": ["Fri, 19 Jun␣
↪2015 04:10:33 GMT"], "access-control-allow-origin": ["*"], "content-type": [
↪"application/json"]}, "status": {"message": "OK", "code": 200}, "url": "https://
↪httpbin.org/get"}, "recorded_at": "2015-06-19T04:10:33"}], "recorded_with":
↪"betamax/0.4.1"}
```

Now, each subsequent time that we run that script, we will use the recorded interaction instead of talking to the internet over and over again.

Recording More Complex Cassettes

Most times we cannot isolate our tests to a single request at a time, so we'll have cassettes that make multiple requests. Betamax can handle these with ease, let's take a look at an example.

```
import betamax
from betamax_serializers import pretty_json
import requests

CASSETTE_LIBRARY_DIR = 'examples/cassettes/'


def main():
    session = requests.Session()
    betamax.Betamax.register_serializer(pretty_json.PrettyJSONSerializer)
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('more-complicated-cassettes',
                               serialize_with='prettyjson'):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})


if __name__ == '__main__':
    main()
```

Before we run this example, we have to install a new package: `betamax-serializers`, e.g., `pip install betamax-serializers`.

If we now run our new example, we'll see a new file appear in our `examples/cassettes/` directory named `more-complicated-cassettes.json`. This cassette will be much larger as a result of making 3 requests and receiving 3 responses. You'll also notice that we imported `betamax_serializers.pretty_json` and called *`register_serializer()`* with `PrettyJSONSerializer`. Then we added a keyword argument to our invocation of *`use_cassette()`*, `serialize_with='prettyjson'`. `PrettyJSONSerializer` is a class provided by the `betamax-serializers` package on PyPI that can serialize and deserialize cassette data into JSON while allowing it to be easily human readable and pretty. Let's see the results:

```
{
  "http_interactions": [
    {
      "recorded_at": "2015-06-21T19:22:54",
      "request": {
        "body": {
          "encoding": "utf-8",
          "string": ""
        },
        "headers": {
          "Accept": [
            "*/*"
```

```
          ],
          "Accept-Encoding": [
            "gzip, deflate"
          ],
          "Connection": [
            "keep-alive"
          ],
          "User-Agent": [
            "python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0"
          ]
        },
        "method": "GET",
        "uri": "https://httpbin.org/get"
      },
      "response": {
        "body": {
          "encoding": null,
          "string": "{\n  \"args\": {}, \n  \"headers\": {\n    \"Accept\": \"*/*\",
↪\n    \"Accept-Encoding\": \"gzip, deflate\", \n    \"Host\": \"httpbin.org\", \n
↪ \"User-Agent\": \"python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0\"\n  }, \n  \
↪"origin\": \"127.0.0.1\", \n  \"url\": \"https://httpbin.org/get\"\n}\n"
        },
        "headers": {
          "access-control-allow-credentials": [
            "true"
          ],
          "access-control-allow-origin": [
            "*"
          ],
          "connection": [
            "keep-alive"
          ],
          "content-length": [
            "265"
          ],
          "content-type": [
            "application/json"
          ],
          "date": [
            "Sun, 21 Jun 2015 19:22:54 GMT"
          ],
          "server": [
            "nginx"
          ]
        },
        "status": {
          "code": 200,
          "message": "OK"
        },
        "url": "https://httpbin.org/get"
      }
    },
    {
      "recorded_at": "2015-06-21T19:22:54",
      "request": {
        "body": {
          "encoding": "utf-8",
          "string": "{\"some-attribute\": \"some-value\"}"
```

```
        },
        "headers": {
          "Accept": [
            "*/*"
          ],
          "Accept-Encoding": [
            "gzip, deflate"
          ],
          "Connection": [
            "keep-alive"
          ],
          "Content-Length": [
            "32"
          ],
          "Content-Type": [
            "application/json"
          ],
          "User-Agent": [
            "python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0"
          ]
        },
        "method": "POST",
        "uri": "https://httpbin.org/post?id=20"
      },
      "response": {
        "body": {
          "encoding": null,
          "string": "{\n  \"args\": {\n    \"id\": \"20\"\n  }, \n  \"data\": \"{\\\
↪"some-attribute\\\": \\\"some-value\\\"}\", \n  \"files\": {}, \n  \"form\": {}, \n␣
↪ \"headers\": {\n    \"Accept\": \"*/*\", \n    \"Accept-Encoding\": \"gzip,␣
↪deflate\", \n    \"Content-Length\": \"32\", \n    \"Content-Type\": \"application/
↪json\", \n    \"Host\": \"httpbin.org\", \n    \"User-Agent\": \"python-requests/2.
↪7.0 CPython/2.7.9 Darwin/14.1.0\"\n  }, \n  \"json\": {\n    \"some-attribute\": \
↪"some-value\"\n  }, \n  \"origin\": \"127.0.0.1\", \n  \"url\": \"https://httpbin.
↪org/post?id=20\"\n}\n"
        },
        "headers": {
          "access-control-allow-credentials": [
            "true"
          ],
          "access-control-allow-origin": [
            "*"
          ],
          "connection": [
            "keep-alive"
          ],
          "content-length": [
            "495"
          ],
          "content-type": [
            "application/json"
          ],
          "date": [
            "Sun, 21 Jun 2015 19:22:54 GMT"
          ],
          "server": [
            "nginx"
          ]
```

```
        },
        "status": {
          "code": 200,
          "message": "OK"
        },
        "url": "https://httpbin.org/post?id=20"
      }
    },
    {
      "recorded_at": "2015-06-21T19:22:54",
      "request": {
        "body": {
          "encoding": "utf-8",
          "string": ""
        },
        "headers": {
          "Accept": [
            "*/*"
          ],
          "Accept-Encoding": [
            "gzip, deflate"
          ],
          "Connection": [
            "keep-alive"
          ],
          "User-Agent": [
            "python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0"
          ]
        },
        "method": "GET",
        "uri": "https://httpbin.org/get?id=20"
      },
      "response": {
        "body": {
          "encoding": null,
          "string": "{\n  \"args\": {\n    \"id\": \"20\"\n  }, \n  \"headers\": {\n
↪  \"Accept\": \"*/*\", \n    \"Accept-Encoding\": \"gzip, deflate\", \n    \"Host\
↪": \"httpbin.org\", \n    \"User-Agent\": \"python-requests/2.7.0 CPython/2.7.9
↪Darwin/14.1.0\"\n  }, \n  \"origin\": \"127.0.0.1\", \n  \"url\": \"https://httpbin.
↪org/get?id=20\"\n}\n"
        },
        "headers": {
          "access-control-allow-credentials": [
            "true"
          ],
          "access-control-allow-origin": [
            "*"
          ],
          "connection": [
            "keep-alive"
          ],
          "content-length": [
            "289"
          ],
          "content-type": [
            "application/json"
          ],
          "date": [
```

```
            "Sun, 21 Jun 2015 19:22:54 GMT"
          ],
          "server": [
            "nginx"
          ]
        },
        "status": {
          "code": 200,
          "message": "OK"
        },
        "url": "https://httpbin.org/get?id=20"
      }
    }
  ],
  "recorded_with": "betamax/0.4.2"
}
```

This makes the cassette easy to read and helps us recognize that requests and responses are paired together. We'll explore cassettes more a bit later.

Long Term Usage Patterns

Now that we've covered the basics in *Getting Started*, let's look at some patterns and problems we might encounter when using Betamax over a period of months instead of minutes.

Adding New Requests to a Cassette

Let's reuse an example. Specifically let's reuse our `examples/more_complicated_cassettes.py` example.

```
import betamax
from betamax_serializers import pretty_json
import requests

CASSETTE_LIBRARY_DIR = 'examples/cassettes/'


def main():
    session = requests.Session()
    betamax.Betamax.register_serializer(pretty_json.PrettyJSONSerializer)
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('more-complicated-cassettes',
                               serialize_with='prettyjson'):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})


if __name__ == '__main__':
    main()
```

Let's add a new `POST` request in there:

```
session.post('https://httpbin.org/post',
             params={'id': '20'},
             json={'some-other-attribute': 'some-other-value'})
```

If we run this cassette now, we should expect to see that there was an exception because Betamax couldn't find a matching request for it. We expect this because the post requests have two completely different bodies, right? Right. The problem you'll find is that by default Betamax **only** matches on the URI and the Method. So Betamax will find a matching request/response pair for `("POST", "https://httpbin.org/post?id=20")` and reuse it. So now we need to update how we use Betamax so it will match using the `body` as well:

```
import betamax
from betamax_serializers import pretty_json
import requests

CASSETTE_LIBRARY_DIR = 'examples/cassettes/'


def main():
    session = requests.Session()
    betamax.Betamax.register_serializer(pretty_json.PrettyJSONSerializer)
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )
    matchers = ['method', 'uri', 'body']

    with recorder.use_cassette('more-complicated-cassettes',
                               serialize_with='prettyjson',
                               match_requests_on=matchers):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-other-attribute': 'some-other-value'})


if __name__ == '__main__':
    main()
```

Now when we run that we should see something like this:

```
Traceback (most recent call last):
  File "examples/more_complicated_cassettes_2.py", line 30, in <module>
    main()
  File "examples/more_complicated_cassettes_2.py", line 26, in main
    json={'some-other-attribute': 'some-other-value'})
  File ".../lib/python2.7/site-packages/requests/sessions.py", line 508, in post
    return self.request('POST', url, data=data, json=json, **kwargs)
  File ".../lib/python2.7/site-packages/requests/sessions.py", line 465, in request
    resp = self.send(prep, **send_kwargs)
  File ".../lib/python2.7/site-packages/requests/sessions.py", line 573, in send
    r = adapter.send(request, **kwargs)
  File ".../lib/python2.7/site-packages/betamax/adapter.py", line 91, in send
    self.cassette))
```

```
betamax.exceptions.BetamaxError: A request was made that could not be handled.

A request was made to https://httpbin.org/post?id=20 that could not be found in more-
↪complicated-cassettes.

The settings on the cassette are:

    - record_mode: once
    - match_options ['method', 'uri', 'body'].
```

This is what we do expect to see. So, how do we fix it?

We have a few options to fix it.

Option 1: Re-recording the Cassette

One of the easiest ways to fix this situation is to simply remove the cassette that was recorded and run the script again. This will recreate the cassette and subsequent runs will work just fine.

To be clear, we're advocating for this option that the user do:

```
$ rm examples/cassettes/{{ cassette-name }}
```

This is the favorable option if you don't foresee yourself needing to add new interactions often.

Option 2: Changing the Record Mode

A different way would be to update the recording mode used by Betamax. We would update the line in our file that currently reads:

```
with recorder.use_cassette('more-complicated-cassettes',
                           serialize_with='prettyjson',
                           match_requests_on=matchers):
```

to add one more parameter to the call to *use_cassette()*. We want to use the `record` parameter to tell Betamax to use either the `new_episodes` or `all` modes. Which you choose depends on your use case.

`new_episodes` will only record new request/response interactions that Betamax sees. `all` will just re-record every interaction every time. In our example, we'll use `new_episodes` so our code now looks like:

```
with recorder.use_cassette('more-complicated-cassettes',
                           serialize_with='prettyjson',
                           match_requests_on=matchers,
                           record='new_episodes'):
```

Known Issues

Tests Periodically Slow Down

Description:

Requests checks if it should use or bypass proxies using the standard library function `proxy_bypass`. This has been known to cause slow downs when using Requests and can cause your recorded requests to slow down as well.

Betamax presently has no way to prevent this from being called as it operates at a lower level in Requests than is necessary.

Workarounds:

- Mock gethostbyname method from socket library, to force a localhost setting, e.g.,

```
import socket
socket.gethostbyname = lambda x: '127.0.0.1'
```

- Set `trust_env` to `False` on the session used with Betamax. This will prevent Requests from checking for proxies and whether it needs bypass them.

Related bugs:

- https://github.com/sigmavirus24/betamax/issues/96
- https://github.com/kennethreitz/requests/issues/2988

Configuring Betamax

By now you've seen examples where we pass a great deal of keyword arguments to *use_cassette()*. You have also seen that we used *betamax.Betamax.configure()*. In this section, we'll go into a deep description of the different approaches and why you might pick one over the other.

Global Configuration

Admittedly, I am not too proud of my decision to borrow this design from VCR, but I did and I use it and it isn't entirely terrible. (Note: I do hope to come up with an elegant way to redesign it for v1.0.0 but that's a long way off.)

The best way to configure Betamax globally is by using *betamax.Betamax.configure()*. This returns a *betamax.configure.Configuration* instance. This instance can be used as a context manager in order to make the usage look more like VCR's way of configuring the library. For example, in VCR, you might do

```
VCR.configure do |config|
  config.cassette_library_dir = 'examples/cassettes'
  config.default_cassette_options[:record] = :none
  # ...
end
```

Where as with Betamax you might do

```
from betamax import Betamax

with Betamax.configure() as config:
    config.cassette_library_dir = 'examples/cassettes'
    config.default_cassette_options['record_mode'] = 'none'
```

Alternatively, since the object returned is really just an object and does not do anything special as a context manager, you could just as easily do

```
from betamax import Betamax

config = Betamax.configure()
config.cassette_library_dir = 'examples/cassettes'
config.default_cassette_options['record_mode'] = 'none'
```

We'll now move on to specific use-cases when configuring Betamax. We'll exclude the portion of each example where we create a *Configuration* instance.

Setting the Directory in which Betamax Should Store Cassette Files

Each and every time we use Betamax we need to tell it where to store (and discover) cassette files. By default we do this by setting the `cassette_library_dir` attribute on our `config` object, e.g.,

```
config.cassette_library_dir = 'tests/integration/cassettes'
```

Note that these paths are relative to what Python thinks is the current working directory. Wherever you run your tests from, write the path to be relative to that directory.

Setting Default Cassette Options

Cassettes have default options used by Betamax if none are set. For example,

- The default record mode is `once`.
- The default matchers used are `method` and `uri`.
- Cassettes do **not** preserve the exact body bytes by default.

These can all be configured as you please. For example, if you want to change the default matchers and preserve exact body bytes, you would do

```
config.default_cassette_options['match_requests_on'] = [
    'method',
    'uri',
    'headers',
]
config.preserve_exact_body_bytes = True
```

Filtering Sensitive Data

It's unlikely that you'll want to record an interaction that will not require authentication. For this we can define placeholders in our cassettes. Let's use a very real example.

Let's say that you want to get your user data from GitHub using Requests. You might have code that looks like this:

```
def me(username, password, session):
    r = session.get('https://api.github.com/user', auth=(username, password))
    r.raise_for_status()
    return r.json()
```

You would test this something like:

```
import os

import betamax
import requests

from my_module import me

session = requests.Session()
recorder = betamax.Betamax(session)
```

```
username = os.environ.get('USERNAME', 'testuser')
password = os.environ.get('PASSWORD', 'testpassword')

with recorder.use_cassette('test-me'):
    json = me(username, password, session)
    # assertions about the JSON returned
```

The problem is that now your username and password will be recorded in the cassette which you don't then want to push to your version control. How can we prevent that from happening?

```
import base64

username = os.environ.get('USERNAME', 'testuser')
password = os.environ.get('PASSWORD', 'testpassword')
config.define_cassette_placeholder(
    '<GITHUB-AUTH>',
    base64.b64encode(
        '{0}:{1}'.format(username, password).encode('utf-8')
    )
)
```

Note: Obviously you can refactor this a bit so you can pull those environment variables out in only one place, but I'd rather be clear than not here.

The first time you run the test script you would invoke your tests like so:

```
$ USERNAME='my-real-username' PASSWORD='supersecretep@55w0rd' \
  python test_script.py
```

Future runs of the script could simply be run without those environment variables, e.g.,

```
$ python test_script.py
```

This means that you can run these tests on a service like Travis-CI without providing credentials.

In the event that you can not anticipate what you will need to filter out, version 0.7.0 of Betamax adds `before_record` and `before_playback` hooks. These two hooks both will pass the `Interaction` and `Cassette` to the function provided. An example callback would look like:

```
def hook(interaction, cassette):
    pass
```

You would then register this callback:

```
# Either
config.before_record(callback=hook)
# Or
config.before_playback(callback=hook)
```

You can register callables for both hooks. If you wish to ignore an interaction and prevent it from being recorded or replayed, you can call the `ignore()`. You also have full access to all of the methods and attributes on an instance of an Interaction. This will allow you to inspect the response produced by the interaction and then modify it. Let's say, for example, that you are talking to an API that grants authorization tokens on a specific request. In this example, you might authenticate initially using a username and password and then use a token after authenticating. You want, however, for the token to be kept secret. In that case you might configure Betamax to replace the username and password, e.g.,

```
config.define_cassette_placeholder('<USERNAME>', username)
config.define_cassette_placeholder('<PASSWORD>', password)
```

And you would also write a function that, prior to recording, finds the token, saves it, and obscures it from the recorded version of the cassette:

```
from betamax.cassette import cassette


def sanitize_token(interaction, current_cassette):
    # Exit early if the request did not return 200 OK because that's the
    # only time we want to look for Authorization-Token headers
    if interaction.data['response']['status']['code'] != 200:
        return

    headers = interaction.data['response']['headers']
    token = headers.get('Authorization-Token')
    # If there was no token header in the response, exit
    if token is None:
        return

    # Otherwise, create a new placeholder so that when cassette is saved,
    # Betamax will replace the token with our placeholder.
    current_cassette.placeholders.append(
        cassette.Placeholder(placeholder='<AUTH_TOKEN>', replace=token)
    )
```

This will dynamically create a placeholder for that cassette only. Once we have our hook, we need merely register it like so:

```
config.before_record(callback=sanitize_token)
```

And we no longer need to worry about leaking sensitive data.

Setting default serializer

If you want to use a specific serializer for every cassette, you can set `serialize_with` as a default cassette option. For example, if you wanted to use the `prettyjson` serializer for every cassette you would do:

```
config.default_cassette_options['serialize_with'] = 'prettyjson'
```

Per-Use Configuration

Each time you create a *Betamax* instance or use *use_cassette()*, you can pass some of the options from above.

Setting the Directory in which Betamax Should Store Cassette Files

When using per-use configuration of Betamax, you can specify the cassette directory when you instantiate a *Betamax* object:

```
session = requests.Session()
recorder = betamax.Betamax(session,
                           cassette_library_dir='tests/cassettes/')
```

Setting Default Cassette Options

You can also set default cassette options when instantiating a *Betamax* object:

```
session = requests.Session()
recorder = betamax.Betamax(session, default_cassette_options={
    'record_mode': 'once',
    'match_requests_on': ['method', 'uri', 'headers'],
    'preserve_exact_body_bytes': True
})
```

You can also set the above when calling *use_cassette()*:

```
session = requests.Session()
recorder = betamax.Betamax(session)
with recorder.use_cassette('cassette-name',
                           preserve_exact_body_bytes=True,
                           match_requests_on=['method', 'uri', 'headers'],
                           record='once'):
    session.get('https://httpbin.org/get')
```

Filtering Sensitive Data

Filtering sensitive data on a per-usage basis is the only difficult (or perhaps, less convenient) case. Cassette placeholders are part of the default cassette options, so we'll set this value similarly to how we set the other default cassette options, the catch is that placeholders have a specific structure. Placeholders are stored as a list of dictionaries. Let's use our example above and convert it.

```
import base64

username = os.environ.get('USERNAME', 'testuser')
password = os.environ.get('PASSWORD', 'testpassword')
session = requests.Session()

recorder = betamax.Betamax(session, default_cassette_options={
    'placeholders': [{
        'placeholder': '<GITHUB-AUTH>',
        'replace': base64.b64encode(
            '{0}:{1}'.format(username, password).encode('utf-8')
        ),
    }]
})
```

Note that what we passed as our first argument is assigned to the `'placeholder'` key while the value we're replacing is assigned to the `'replace'` key.

This isn't the typical way that people filter sensitive data because they tend to want to do it globally.

Mixing and Matching

It's not uncommon to mix and match configuration methodologies. I do this in github3.py. I use global configuration to filter sensitive data and set defaults based on the environment the tests are running in. On Travis-CI, the record mode is set to `'none'`. I also set how we match requests and when we preserve exact body bytes on a per-use basis.

Record Modes

Betamax, like VCR, has four modes that it can use to record cassettes:

- `'all'`
- `'new_episodes'`
- `'none'`
- `'once'`

You can only ever use one record mode. Below are explanations and examples of each record mode. The explanations are blatantly taken from VCR's own Record Modes documentation.

All

The `'all'` record mode will:

- Record new interactions.
- Never replay previously recorded interactions.

This can be temporarily used to force VCR to re-record a cassette (i.e., to ensure the responses are not out of date) or can be used when you simply want to log all HTTP requests.

Given our file, `examples/record_modes/all/example.py`,

```
import betamax
import requests

CASSETTE_LIBRARY_DIR = 'examples/record_modes/all/'


def main():
    session = requests.Session()
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('all-example', record='all'):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})


if __name__ == '__main__':
    main()
```

Every time we run it, our cassette (`examples/record_modes/all/all-example.json`) will be updated with new values.

New Episodes

The `'new_episodes'` record mode will:

- Record new interactions.

- Replay previously recorded interactions.

It is similar to the `'once'` record mode, but will always record new interactions, even if you have an existing recorded one that is similar (but not identical, based on the :match_request_on option).

Given our file, `examples/record_modes/new_episodes/example_original.py`, with which we have already recorded `examples/record_modes/new_episodes/new-episodes-example.json`

```
import betamax
import requests

CASSETTE_LIBRARY_DIR = 'examples/record_modes/new_episodes/'


def main():
    session = requests.Session()
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('new-episodes-example', record='new_episodes'):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})


if __name__ == '__main__':
    main()
```

If we then run `examples/record_modes/new_episodes/example_updated.py`

```
import betamax
import requests

CASSETTE_LIBRARY_DIR = 'examples/record_modes/new_episodes/'


def main():
    session = requests.Session()
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('new-episodes-example', record='new_episodes'):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})
        session.get('https://httpbin.org/get', params={'id': '40'})


if __name__ == '__main__':
    main()
```

The new request at the end of the file will be added to the cassette without updating the other interactions that were already recorded.

None

The `'none'` record mode will:

- Replay previously recorded interactions.
- Cause an error to be raised for any new requests.

This is useful when your code makes potentially dangerous HTTP requests. The `'none'` record mode guarantees that no new HTTP requests will be made.

Given our file, `examples/record_modes/none/example_original.py`, with a cassette that already has interactions recorded in `examples/record_modes/none/none-example.json`

```
import betamax
import requests

CASSETTE_LIBRARY_DIR = 'examples/record_modes/none/'


def main():
    session = requests.Session()
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('none-example', record='none'):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})


if __name__ == '__main__':
    main()
```

If we then run `examples/record_modes/none/example_updated.py`

```
import betamax
import requests

CASSETTE_LIBRARY_DIR = 'examples/record_modes/none/'


def main():
    session = requests.Session()
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('none-example', record='none'):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})
        session.get('https://httpbin.org/get', params={'id': '40'})
```

```
if __name__ == '__main__':
    main()
```

We'll see an exception indicating that new interactions were prevented:

```
Traceback (most recent call last):
  File "examples/record_modes/none/example_updated.py", line 23, in <module>
    main()
  File "examples/record_modes/none/example_updated.py", line 19, in main
    session.get('https://httpbin.org/get', params={'id': '40'})
  File "/usr/local/lib/python2.7/site-packages/requests/sessions.py", line 477, in get
    return self.request('GET', url, **kwargs)
  File "/usr/local/lib/python2.7/site-packages/requests/sessions.py", line 465, in␣
↪request
    resp = self.send(prep, **send_kwargs)
  File "/usr/local/lib/python2.7/site-packages/requests/sessions.py", line 573, in␣
↪send
    r = adapter.send(request, **kwargs)
  File "/usr/local/lib/python2.7/site-packages/betamax/adapter.py", line 91, in send
    self.cassette))
betamax.exceptions.BetamaxError: A request was made that could not be handled.

A request was made to https://httpbin.org/get?id=40 that could not be found in none-
↪example.

The settings on the cassette are:

    - record_mode: none
    - match_options ['method', 'uri'].
```

Once

The `'once'` record mode will:

- Replay previously recorded interactions.
- Record new interactions if there is no cassette file.
- Cause an error to be raised for new requests if there is a cassette file.

It is similar to the `'new_episodes'` record mode, but will prevent new, unexpected requests from being made (i.e. because the request URI changed or whatever).

`'once'` is the default record mode, used when you do not set one.

If we have a file, `examples/record_modes/once/example_original.py`,

```
import betamax
import requests

CASSETTE_LIBRARY_DIR = 'examples/record_modes/once/'


def main():
    session = requests.Session()
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )
```

```
    with recorder.use_cassette('once-example', record='once'):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})


if __name__ == '__main__':
    main()
```

And we run it, we'll see a cassette named `examples/record_modes/once/once-example.json` has been created.

If we then run `examples/record_modes/once/example_updated.py`,

```
import betamax
import requests

CASSETTE_LIBRARY_DIR = 'examples/record_modes/once/'


def main():
    session = requests.Session()
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('once-example', record='once'):
        session.get('https://httpbin.org/get')
        session.post('https://httpbin.org/post',
                     params={'id': '20'},
                     json={'some-attribute': 'some-value'})
        session.get('https://httpbin.org/get', params={'id': '20'})
        session.get('https://httpbin.org/get', params={'id': '40'})


if __name__ == '__main__':
    main()
```

We'll see an exception similar to the one we see when using the `'none'` record mode.

```
Traceback (most recent call last):
  File "examples/record_modes/once/example_updated.py", line 23, in <module>
    main()
  File "examples/record_modes/once/example_updated.py", line 19, in main
    session.get('https://httpbin.org/get', params={'id': '40'})
  File "/usr/local/lib/python2.7/site-packages/requests/sessions.py", line 477, in get
    return self.request('GET', url, **kwargs)
  File "/usr/local/lib/python2.7/site-packages/requests/sessions.py", line 465, in␣
↪request
    resp = self.send(prep, **send_kwargs)
  File "/usr/local/lib/python2.7/site-packages/requests/sessions.py", line 573, in␣
↪send
    r = adapter.send(request, **kwargs)
  File "/usr/local/lib/python2.7/site-packages/betamax/adapter.py", line 91, in send
    self.cassette))
```

```
betamax.exceptions.BetamaxError: A request was made that could not be handled.

A request was made to https://httpbin.org/get?id=40 that could not be found in none-
↪example.

The settings on the cassette are:

    - record_mode: once
    - match_options ['method', 'uri'].
```

Third-Party Packages

Betamax was created to be a very close imitation of VCR. As such, it has the default set of request matchers and a subset of the supported cassette serializers for VCR.

As part of my own usage of Betamax, and supporting other people's usage of Betamax, I've created (and maintain) two third party packages that provide extra request matchers and cassette serializers.

- betamax-matchers
- betamax-serializers

For simplicity, those modules will be documented here instead of on their own documentation sites.

Request Matchers

There are three third-party request matchers provided by the betamax-matchers package:

- `URLEncodedBodyMatcher`, `'form-urlencoded-body'`
- `JSONBodyMatcher`, `'json-body'`
- `MultipartFormDataBodyMatcher`, `'multipart-form-data-body'`

In order to use any of these we have to register them with Betamax. Below we will register all three but you do not need to do that if you only need to use one:

```
import betamax
from betamax_matchers import form_urlencoded
from betamax_matchers import json_body
from betamax_matchers import multipart

betamax.Betamax.register_request_matcher(
    form_urlencoded.URLEncodedBodyMatcher
    )
betamax.Betamax.register_request_matcher(
    json_body.JSONBodyMatcher
    )
betamax.Betamax.register_request_matcher(
    multipart.MultipartFormDataBodyMatcher
    )
```

All of these classes inherit from *`betamax.BaseMatcher`* which means that each needs a name that will be used when specifying what matchers to use with Betamax. I have noted those next to the class name for each matcher above. Let's use the JSON body matcher in an example though:

```
import betamax
from betamax_matchers import json_body
# This example requires at least requests 2.5.0
import requests

betamax.Betamax.register_request_matcher(
    json_body.JSONBodyMatcher
    )


def main():
    session = requests.Session()
    recorder = betamax.Betamax(session, cassette_library_dir='.')
    url = 'https://httpbin.org/post'
    json_data = {'key': 'value',
                 'other-key': 'other-value',
                 'yet-another-key': 'yet-another-value'}
    matchers = ['method', 'uri', 'json-body']

    with recorder.use_cassette('json-body-example', match_requests_on=matchers):
        r = session.post(url, json=json_data)


if __name__ == '__main__':
    main()
```

If we ran that request without those matcher with hash seed randomization, then we would occasionally receive exceptions that a request could not be matched. That is because dictionaries are not inherently ordered so the body string of the request can change and be any of the following:

```
{"key": "value", "other-key": "other-value", "yet-another-key":
"yet-another-value"}
```

```
{"key": "value", "yet-another-key": "yet-another-value", "other-key":
"other-value"}
```

```
{"other-key": "other-value", "yet-another-key": "yet-another-value",
"key": "value"}
```

```
{"yet-another-key": "yet-another-value", "key": "value", "other-key":
"other-value"}
```

```
{"yet-another-key": "yet-another-value", "other-key": "other-value",
"key": "value"}
```

```
{"other-key": "other-value", "key": "value", "yet-another-key":
"yet-another-value"}
```

But using the `'json-body'` matcher, the matcher will parse the request and compare python dictionaries instead of python strings. That will completely bypass the issues introduced by hash randomization. I use this matcher extensively in github3.py's tests.

Cassette Serializers

By default, Betamax only comes with the JSON serializer. betamax-serializers provides extra serializer classes that users have contributed.

For example, as we've seen elsewhere in our documentation, the default JSON serializer does not create beautiful or easy to read cassettes. As a substitute for that, we have the `PrettyJSONSerializer` that does that for you.

```
from betamax import Betamax
from betamax_serializers import pretty_json

import requests

Betamax.register_serializer(pretty_json.PrettyJSONSerializer)

session = requests.Session()
recorder = Betamax(session)
with recorder.use_cassette('testpretty', serialize_with='prettyjson'):
    session.request(method=method, url=url, ...)
```

This will give us a pretty-printed cassette like:

```
{
  "http_interactions": [
    {
      "recorded_at": "2015-06-21T19:22:54",
      "request": {
        "body": {
          "encoding": "utf-8",
          "string": ""
        },
        "headers": {
          "Accept": [
            "*/*"
          ],
          "Accept-Encoding": [
            "gzip, deflate"
          ],
          "Connection": [
            "keep-alive"
          ],
          "User-Agent": [
            "python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0"
          ]
        },
        "method": "GET",
        "uri": "https://httpbin.org/get"
      },
      "response": {
        "body": {
          "encoding": null,
          "string": "{\n  \"args\": {}, \n  \"headers\": {\n    \"Accept\": \"*/*\",␣
↪\n    \"Accept-Encoding\": \"gzip, deflate\", \n    \"Host\": \"httpbin.org\", \n␣
↪ \"User-Agent\": \"python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0\"\n  }, \n  \
↪"origin\": \"127.0.0.1\", \n  \"url\": \"https://httpbin.org/get\"\n}\n"
        },
        "headers": {
          "access-control-allow-credentials": [
            "true"
```

```
          ],
          "access-control-allow-origin": [
            "*"
          ],
          "connection": [
            "keep-alive"
          ],
          "content-length": [
            "265"
          ],
          "content-type": [
            "application/json"
          ],
          "date": [
            "Sun, 21 Jun 2015 19:22:54 GMT"
          ],
          "server": [
            "nginx"
          ]
        },
        "status": {
          "code": 200,
          "message": "OK"
        },
        "url": "https://httpbin.org/get"
      }
    },
    {
      "recorded_at": "2015-06-21T19:22:54",
      "request": {
        "body": {
          "encoding": "utf-8",
          "string": "{\"some-attribute\": \"some-value\"}"
        },
        "headers": {
          "Accept": [
            "*/*"
          ],
          "Accept-Encoding": [
            "gzip, deflate"
          ],
          "Connection": [
            "keep-alive"
          ],
          "Content-Length": [
            "32"
          ],
          "Content-Type": [
            "application/json"
          ],
          "User-Agent": [
            "python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0"
          ]
        },
        "method": "POST",
        "uri": "https://httpbin.org/post?id=20"
      },
      "response": {
```

```
        "body": {
          "encoding": null,
          "string": "{\n  \"args\": {\n    \"id\": \"20\"\n  }, \n  \"data\": \"{\\\
↪"some-attribute\\\": \\\"some-value\\\"}\", \n  \"files\": {}, \n  \"form\": {}, \n␣
↪ \"headers\": {\n    \"Accept\": \"*/*\", \n    \"Accept-Encoding\": \"gzip,␣
↪deflate\", \n    \"Content-Length\": \"32\", \n    \"Content-Type\": \"application/
↪json\", \n    \"Host\": \"httpbin.org\", \n    \"User-Agent\": \"python-requests/2.
↪7.0 CPython/2.7.9 Darwin/14.1.0\"\n  }, \n  \"json\": {\n    \"some-attribute\": \
↪"some-value\"\n  }, \n  \"origin\": \"127.0.0.1\", \n  \"url\": \"https://httpbin.
↪org/post?id=20\"\n}\n"
        },
        "headers": {
          "access-control-allow-credentials": [
            "true"
          ],
          "access-control-allow-origin": [
            "*"
          ],
          "connection": [
            "keep-alive"
          ],
          "content-length": [
            "495"
          ],
          "content-type": [
            "application/json"
          ],
          "date": [
            "Sun, 21 Jun 2015 19:22:54 GMT"
          ],
          "server": [
            "nginx"
          ]
        },
        "status": {
          "code": 200,
          "message": "OK"
        },
        "url": "https://httpbin.org/post?id=20"
      }
    },
    {
      "recorded_at": "2015-06-21T19:22:54",
      "request": {
        "body": {
          "encoding": "utf-8",
          "string": ""
        },
        "headers": {
          "Accept": [
            "*/*"
          ],
          "Accept-Encoding": [
            "gzip, deflate"
          ],
          "Connection": [
            "keep-alive"
          ],
```

```
          "User-Agent": [
            "python-requests/2.7.0 CPython/2.7.9 Darwin/14.1.0"
          ]
        },
        "method": "GET",
        "uri": "https://httpbin.org/get?id=20"
      },
      "response": {
        "body": {
          "encoding": null,
          "string": "{\n  \"args\": {\n    \"id\": \"20\"\n  }, \n  \"headers\": {\n
↪  \"Accept\": \"*/*\", \n    \"Accept-Encoding\": \"gzip, deflate\", \n    \"Host\
↪": \"httpbin.org\", \n    \"User-Agent\": \"python-requests/2.7.0 CPython/2.7.9
↪Darwin/14.1.0\"\n  }, \n  \"origin\": \"127.0.0.1\", \n  \"url\": \"https://httpbin.
↪org/get?id=20\"\n}\n"
        },
        "headers": {
          "access-control-allow-credentials": [
            "true"
          ],
          "access-control-allow-origin": [
            "*"
          ],
          "connection": [
            "keep-alive"
          ],
          "content-length": [
            "289"
          ],
          "content-type": [
            "application/json"
          ],
          "date": [
            "Sun, 21 Jun 2015 19:22:54 GMT"
          ],
          "server": [
            "nginx"
          ]
        },
        "status": {
          "code": 200,
          "message": "OK"
        },
        "url": "https://httpbin.org/get?id=20"
      }
    }
  ],
  "recorded_with": "betamax/0.4.2"
}
```


Usage Patterns

Below are suggested patterns for using Betamax efficiently.

Configuring Betamax in py.test's conftest.py

Betamax and github3.py (the project which instigated the creation of Betamax) both utilize py.test and its feature of configuring how the tests run with `conftest.py`[1]. One pattern that I have found useful is to include this in your `conftest.py` file:

```
import betamax

with betamax.Betamax.configure() as config:
    config.cassette_library_dir = 'tests/cassettes/'
```

This configures your cassette directory for all of your tests. If you do not check your cassettes into your version control system, then you can also add:

```
import os

if not os.path.exists('tests/cassettes'):
    os.makedirs('tests/cassettes')
```

An Example from github3.py

You can configure other aspects of Betamax via the `conftest.py` file. For example, in github3.py, I do the following:

```
import os

record_mode = 'none' if os.environ.get('TRAVIS_GH3') else 'once'

with betamax.Betamax.configure() as config:
    config.cassette_library_dir = 'tests/cassettes/'
    config.default_cassette_options['record_mode'] = record_mode
    config.define_cassette_placeholder(
        '<AUTH_TOKEN>',
        os.environ.get('GH_AUTH', 'x' * 20)
    )
```

In essence, if the tests are being run on Travis CI, then we want to make sure to not try to record new cassettes or interactions. We also, want to ensure we're authenticated when possible but that we do not leave our placeholder in the cassettes when they're replayed.

Using Human Readable JSON Cassettes

Using the `PrettyJSONSerializer` provided by the `betamax_serializers` package provides human readable JSON cassettes. Cassettes output in this way make it easy to compare modifications to cassettes to ensure only expected changes are introduced.

While you can use the `serialize_with` option when creating each individual cassette, it is simpler to provide this setting globally. The following example demonstrates how to configure Betamax to use the `PrettyJSONSerializer` for all newly created cassettes:

```
from betamax_serializers import pretty_json
betamax.Betamax.register_serializer(pretty_json.PrettyJSONSerializer)
# ...
config.default_cassette_options['serialize_with'] = 'prettyjson'
```

[1] http://pytest.org/latest/plugins.html

Updating Existing Betamax Cassettes to be Human Readable

If you already have a library of cassettes when applying the previous configuration update, then you will probably want to also update all your existing cassettes into the new human readable format. The following script will help you transform your existing cassettes:

```
import os
import glob
import json
import sys

try:
    cassette_dir = sys.argv[1]
    cassettes = glob.glob(os.path.join(cassette_dir, '*.json'))
except:
    print('Usage: {0} CASSETTE_DIRECTORY'.format(sys.argv[0]))
    sys.exit(1)

for cassette_path in cassettes:
    with open(cassette_path, 'r') as fp:
        data = json.load(fp)
    with open(cassette_path, 'w') as fp:
        json.dump(data, fp, sort_keys=True, indent=2,
                  separators=(',', ': '))
print('Updated {0} cassette{1}.'.format(
    len(cassettes), '' if len(cassettes) == 1 else 's'))
```

Copy and save the above script as `fix_cassettes.py` and then run it like:

```
python fix_cassettes.py PATH_TO_CASSETTE_DIRECTORY
```

If you're not already using a version control system (e.g., git, svn) then it is recommended you make a backup of your cassettes first in the event something goes wrong.

Integrating Betamax with Test Frameworks

It's nice to have a way to integrate libraries you use for testing into your testing frameworks. Having considered this, the authors of and contributors to Betamax have included integrations in the package. Betamax comes with integrations for py.test and unittest. (If you need an integration for another framework, please suggest it and send a patch!)

PyTest Integration

New in version 0.5.0.

Changed in version 0.6.0.

When you install Betamax, it now installs two py.test fixtures by default. To use it in your tests you need only follow the instructions on pytest's documentation. To use the `betamax_session` fixture for an entire class of tests you would do:

```
# tests/test_http_integration.py
import pytest

@pytest.mark.usefixtures('betamax_session')
class TestMyHttpClient:
    def test_get(self, betamax_session):
        betamax_session.get('https://httpbin.org/get')
```

This will generate a cassette name for you, e.g., `tests.test_http_integration.TestMyHttpClient.test_get`. After running this test you would have a cassette file stored in your cassette library directory named `tests.test_http_integration.TestMyHttpClient.test_get.json`. To use this fixture at the module level, you need only do

```
# tests/test_http_integration.py
import pytest

pytest.mark.usefixtures('betamax_session')


class TestMyHttpClient:
    def test_get(self, betamax_session):
        betamax_session.get('https://httpbin.org/get')

class TestMyOtherHttpClient:
    def test_post(self, betamax_session):
        betamax_session.post('https://httpbin.org/post')
```

If you need to customize the recorder object, however, you can instead use the `betamax_recorder` fixture:

```
# tests/test_http_integration.py
import pytest

pytest.mark.usefixtures('betamax_recorder')


class TestMyHttpClient:
    def test_post(self, betamax_recorder):
        betamax_recorder.current_cassette.match_options.add('json-body')
        session = betamax_recorder.session

        session.post('https://httpbin.org/post', json={'foo': 'bar'})
```

Unittest Integration

New in version 0.5.0.

When writing tests with unittest, a common pattern is to either import `unittest.TestCase` or subclass that and use that subclass in your tests. When integrating Betamax with your unittest testsuite, you should do the following:

```
from betamax.fixtures import unittest


class IntegrationTestCase(unittest.BetamaxTestCase):
    # Add the rest of the helper methods you want for your
    # integration tests
```

```
class SpecificTestCase(IntegrationTestCase):
    def test_something(self):
        # Test something
```

The unittest integration provides the following attributes on the test case instance:

- `session` the instance of `BetamaxTestCase.SESSION_CLASS` created for that test.
- `recorder` the instance of *betamax.Betamax* created.

The integration also generates a cassette name from the test case class name and test method. So the cassette generated for the above example would be named `SpecificTestCase.test_something`. To override that behaviour, you need to override the `generate_cassette_name()` method in your subclass.

If you are subclassing `requests.Session` in your application, then it follows that you will want to use that in your tests. To facilitate this, you can set the `SESSION_CLASS` attribute. To give a fuller example, let's say you're changing the default cassette name and you're providing your own session class, your code might look like:

```
from betamax.fixtures import unittest

from myapi import session


class IntegrationTestCase(unittest.BetamaxTestCase):
    # Add the rest of the helper methods you want for your
    # integration tests
    SESSION_CLASS = session.MyApiSession

    def generate_cassette_name(self):
        classname = self.__class__.__name__
        method = self._testMethodName
        return 'integration_{0}_{1}'.format(classname, method)
```

API

class betamax.**Betamax**(*session*, *cassette_library_dir=None*, *default_cassette_options={}*)

This object contains the main API of the request-vcr library.

This object is entirely a context manager so all you have to do is:

```
s = requests.Session()
with Betamax(s) as vcr:
    vcr.use_cassette('example')
    r = s.get('https://httpbin.org/get')
```

Or more concisely, you can do:

```
s = requests.Session()
with Betamax(s).use_cassette('example') as vcr:
    r = s.get('https://httpbin.org/get')
```

This object allows for the user to specify the cassette library directory and default cassette options.

```
s = requests.Session()
with Betamax(s, cassette_library_dir='tests/cassettes') as vcr:
    vcr.use_cassette('example')
```

```
    r = s.get('https://httpbin.org/get')

with Betamax(s, default_cassette_options={
        're_record_interval': 1000
        }) as vcr:
    vcr.use_cassette('example')
    r = s.get('https://httpbin.org/get')
```

betamax_adapter = None
Create a new adapter to replace the existing ones

static configure()
Help to configure the library as a whole.

```
with Betamax.configure() as config:
    config.cassette_library_dir = 'tests/cassettes/'
    config.default_cassette_options['match_options'] = [
        'method', 'uri', 'headers'
        ]
```

current_cassette
Return the cassette that is currently in use.

Returns *Cassette*

http_adapters = None
Store the session's original adapters.

static register_request_matcher(*matcher_class*)
Register a new request matcher.

Parameters **matcher_class** – (required), this must sub-class `BaseMatcher`

static register_serializer(*serializer_class*)
Register a new serializer.

Parameters **matcher_class** – (required), this must sub-class `BaseSerializer`

session = None
Store the requests.Session object being wrapped.

start()
Start recording or replaying interactions.

stop()
Stop recording or replaying interactions.

use_cassette(*cassette_name*, ***kwargs*)
Tell Betamax which cassette you wish to use for the context.

Parameters

- **cassette_name** (*str*) – relative name, without the serialization format, of the cassette you wish Betamax would use
- **serialize_with** (*str*) – the format you want Betamax to serialize the cassette with
- **serialize** (*str*) – DEPRECATED the format you want Betamax to serialize the request and response data to and from

`betamax.decorator.`**use_cassette**(*cassette_name*, *cassette_library_dir=None*, *default_cassette_options={}*, ***use_cassette_kwargs*)
Provide a Betamax-wrapped Session for convenience.

New in version 0.5.0.

This decorator can be used to get a plain Session that has been wrapped in Betamax. For example,

```
from betamax.decorator import use_cassette

@use_cassette('example-decorator', cassette_library_dir='.')
def test_get(session):
    # do things with session
```

Parameters

- **cassette_name** (*str*) – Name of the cassette file in which interactions will be stored.
- **cassette_library_dir** (*str*) – Directory in which cassette files will be stored.
- **default_cassette_options** (*dict*) – Dictionary of default cassette options to set for the cassette used when recording these interactions.
- ****use_cassette_kwargs** – Keyword arguments passed to *use_cassette()*

class betamax.configure.**Configuration**

This object acts as a proxy to configure different parts of Betamax.

You should only ever encounter this object when configuring the library as a whole. For example:

```
with Betamax.configure() as config:
    config.cassette_library_dir = 'tests/cassettes/'
    config.default_cassette_options['record_mode'] = 'once'
    config.default_cassette_options['match_requests_on'] = ['uri']
    config.define_cassette_placeholder('<URI>', 'http://httpbin.org')
    config.preserve_exact_body_bytes = True
```

before_playback(*tag=None*, *callback=None*)

Register a function to call before playing back an interaction.

Example usage:

```
def before_playback(interaction, cassette):
    pass

with Betamax.configure() as config:
    config.before_playback(callback=before_playback)
```

Parameters

- **tag** (*str*) – Limits the interactions passed to the function based on the interaction's tag (currently unsupported).
- **callback** (*callable*) – The function which either accepts just an interaction or an interaction and a cassette and mutates the interaction before returning.

before_record(*tag=None*, *callback=None*)

Register a function to call before recording an interaction.

Example usage:

```
def before_record(interaction, cassette):
    pass
```

```
with Betamax.configure() as config:
    config.before_record(callback=before_record)
```

Parameters

- **tag** (*str*) – Limits the interactions passed to the function based on the interaction's tag (currently unsupported).
- **callback** (*callable*) – The function which either accepts just an interaction or an interaction and a cassette and mutates the interaction before returning.

cassette_library_dir
Retrieve and set the directory to store the cassettes in.

default_cassette_options
Retrieve and set the default cassette options.

The options include:

- match_requests_on
- placeholders
- re_record_interval
- record_mode
- preserve_exact_body_bytes

Other options will be ignored.

define_cassette_placeholder(*placeholder*, *replace*)
Define a placeholder value for some text.

This also will replace the placeholder text with the text you wish it to use when replaying interactions from cassettes.

Parameters

- **placeholder** (*str*) – (required), text to be used as a placeholder
- **replace** (*str*) – (required), text to be replaced or replacing the placeholder

A set of fixtures to integrate Betamax with py.test.

betamax.fixtures.pytest.**betamax_session**(*betamax_recorder*)
Generate a session that has Betamax already installed.

See *betamax_recorder* fixture.

Parameters **betamax_recorder** – A recorder fixture with a configured request session.

Returns An instantiated requests Session wrapped by Betamax.

Minimal unittest.TestCase subclass adding Betamax integration.

class betamax.fixtures.unittest.**BetamaxTestCase**(*methodName='runTest'*)
Betamax integration for unittest.

New in version 0.5.0.

SESSION_CLASS
Class that is a subclass of requests.Session

alias of Session

generate_cassette_name()
Generates a cassette name for the current test.

The default format is "%(classname)s.%(testMethodName)s"

To change the default cassette format, override this method in a subclass.

Returns Cassette name for the current test.

Return type str

setUp()
Betamax-ified setUp fixture.

This will call the superclass' setUp method *first* and then it will create a new `requests.Session` and wrap that in a Betamax object to record it. At the end of `setUp`, it will start recording.

tearDown()
Betamax-ified tearDown fixture.

This will call the superclass' tearDown method *first* and then it will stop recording interactions.

When using Betamax with unittest, you can use the traditional style of Betamax covered in the documentation thoroughly, or you can use your fixture methods, `unittest.TestCase.setUp()` and `unittest.TestCase.tearDown()` to wrap entire tests in Betamax.

Here's how you might use it:

```
from betamax.fixtures import unittest

from myapi import SessionManager


class TestMyApi(unittest.BetamaxTestCase):
    def setUp(self):
        # Call BetamaxTestCase's setUp first to get a session
        super(TestMyApi, self).setUp()

        self.manager = SessionManager(self.session)

    def test_all_users(self):
        """Retrieve all users from the API."""
        for user in self.manager:
            # Make assertions or something
```

Alternatively, if you are subclassing a `requests.Session` to provide extra functionality, you can do something like this:

```
from betamax.fixtures import unittest

from myapi import Session, SessionManager


class TestMyApi(unittest.BetamaxTestCase):
    SESSION_CLASS = Session

    # See above ...
```

Examples

Basic Usage

Let *example.json* be a file in a directory called *cassettes* with the content:

```
{
  "http_interactions": [
    {
      "request": {
        "body": {
          "string": "",
          "encoding": "utf-8"
        },
        "headers": {
          "User-Agent": ["python-requests/v1.2.3"]
        },
        "method": "GET",
        "uri": "https://httpbin.org/get"
      },
      "response": {
        "body": {
          "string": "example body",
          "encoding": "utf-8"
        },
        "headers": {},
        "status": {
          "code": 200,
          "message": "OK"
        },
        "url": "https://httpbin.org/get"
      }
    }
  ],
  "recorded_with": "betamax"
}
```

The following snippet will not raise any exceptions

```
from betamax import Betamax
from requests import Session


s = Session()

with Betamax(s, cassette_library_dir='cassettes') as betamax:
    betamax.use_cassette('example', record='none')
    r = s.get("https://httpbin.org/get")
```

On the other hand, this will raise an exception:

```
from betamax import Betamax
from requests import Session


s = Session()

with Betamax(s, cassette_library_dir='cassettes') as betamax:
    betamax.use_cassette('example', record='none')
    r = s.post("https://httpbin.org/post",
```

```
                data={"key": "value"})
```

Finally, we can also use a decorator in order to simplify things:

```
import unittest

from betamax.decorator import use_cassette

class TestExample(unittest.TestCase):
    @use_cassette('example', cassette_library_dir='cassettes')
    def test_example(self, session):
        session.get('https://httpbin.org/get')


# Or if you're using something like py.test
@use_cassette('example', cassette_library_dir='cassettes')
def test_example_pytest(session):
    session.get('https://httpbin.org/get')
```

Opinions at Work

If you use `requests`'s default `Accept-Encoding` header, servers that support gzip content encoding will return responses that Betamax cannot serialize in a human-readable format. In this event, the cassette will look like this:

```
{
  "http_interactions": [
    {
      "request": {
        "body": {
          "base64_string": "",
          "encoding": "utf-8"
        },
        "headers": {
          "User-Agent": ["python-requests/v1.2.3"]
        },
        "method": "GET",
        "uri": "https://httpbin.org/get"
      },
      "response": {
        "body": {
          "base64_string": "Zm9vIGJhcgo=",
          "encoding": "utf-8"
        },
        "headers": {
          "Content-Encoding": ["gzip"]
        },
        "status": {
          "code": 200,
          "message": "OK"
        },
        "url": "https://httpbin.org/get"
      }
    }
  ],
  "recorded_with": "betamax"
}
```

Forcing bytes to be preserved

You may want to force betamax to preserve the exact bytes in the body of a response (or request) instead of relying on the *opinions held by the library*. In this case you have two ways of telling betamax to do this.

The first, is on a per-cassette basis, like so:

```
from betamax import Betamax
import requests


session = Session()

with Betamax.configure() as config:
    c.cassette_library_dir = '.'

with Betamax(session).use_cassette('some_cassette',
                                   preserve_exact_body_bytes=True):
    r = session.get('http://example.com')
```

On the other hand, you may want to the preserve exact body bytes for all cassettes. In this case, you can do:

```
from betamax import Betamax
import requests


session = Session()

with Betamax.configure() as config:
    c.cassette_library_dir = '.'
    c.preserve_exact_body_bytes = True

with Betamax(session).use_cassette('some_cassette'):
    r = session.get('http://example.com')
```

What is a cassette?

A cassette is a set of recorded interactions serialized to a specific format. Currently the only supported format is JSON. A cassette has a list (or array) of interactions and information about the library that recorded it. This means that the cassette's structure (using JSON) is

```
{
  "http_interactions": [
    // ...
  ],
  "recorded_with": "betamax"
}
```

Each interaction is the object representing the request and response as well as the date it was recorded. The structure of an interaction is

```
{
  "request": {
    // ...
  },
  "response": {
```

```
    // ...
  },
  "recorded_at": "2013-09-28T01:25:38"
}
```

Each request has the body, method, uri, and an object representing the headers. A serialized request looks like:

```
{
  "body": {
    "string": "...",
    "encoding": "utf-8"
  },
  "method": "GET",
  "uri": "http://example.com",
  "headers": {
    // ...
  }
}
```

A serialized response has the status_code, url, and objects representing the headers and the body. A serialized response looks like:

```
{
  "body": {
    "encoding": "utf-8",
    "string": "..."
  },
  "url": "http://example.com",
  "status": {
    "code": 200,
    "message": "OK"
  },
  "headers": {
    // ...
  }
}
```

If you put everything together, you get:

```
{
  "http_interactions": [
    {
      "request": {
        {
          "body": {
            "string": "...",
            "encoding": "utf-8"
          },
          "method": "GET",
          "uri": "http://example.com",
          "headers": {
            // ...
          }
        }
      },
      "response": {
        {
          "body": {
```

```
            "encoding": "utf-8",
            "string": "..."
          },
          "url": "http://example.com",
          "status": {
            "code": 200,
            "message": "OK"
          },
          "headers": {
            // ...
          }
        }
      },
      "recorded_at": "2013-09-28T01:25:38"
    }
  ],
  "recorded_with": "betamax"
}
```

If you were to pretty-print a cassette, this is vaguely what you would see. Keep in mind that since Python does not keep dictionaries ordered, the items may not be in the same order as this example.

Note: **Pro-tip** You can pretty print a cassette like so: `python -m json.tool cassette.json`.

What is a cassette library?

When configuring Betamax, you can choose your own cassette library directory. This is the directory available from the current directory in which you want to store your cassettes.

For example, let's say that you set your cassette library to be `tests/cassettes/`. In that case, when you record a cassette, it will be saved there. To continue the example, let's say you use the following code:

```
from requests import Session
from betamax import Betamax


s = Session()
with Betamax(s, cassette_library_dir='tests/cassettes').use_cassette('example'):
    r = s.get('https://httpbin.org/get')
```

You would then have the following directory structure:

```
.
`-- tests
    `-- cassettes
        `-- example.json
```

Implementation Details

Everything here is an implementation detail and subject to volatile change. I would not rely on anything here for any mission critical code.

Gzip Content-Encoding

By default, requests sets an `Accept-Encoding` header value that includes `gzip` (specifically, unless overridden, requests always sends `Accept-Encoding: gzip, deflate, compress`). When a server supports this and responds with a response that has the `Content-Encoding` header set to `gzip`, `urllib3` automatically decompresses the body for requests. This can only be prevented in the case where the `stream` parameter is set to `True`. Since Betamax refuses to alter the headers on the response object in any way, we force `stream` to be `True` so we can capture the compressed data before it is decompressed. We then properly repopulate the response object so you perceive no difference in the interaction.

To preserve the response exactly as is, we then must `base64` encode the body of the response before saving it to the file object. In other words, whenever a server responds with a compressed body, you will not have a human readable response body. There is, at the present moment, no way to configure this so that this does not happen and because of the way that Betamax works, you can not remove the `Content-Encoding` header to prevent this from happening.

Class Details

class `betamax.cassette.`**`Cassette`**(*cassette_name*, *serialization_format*, ***kwargs*)

`cassette_name` = **None**
Short name of the cassette

`earliest_recorded_date`
The earliest date of all of the interactions this cassette.

`find_match`(*request*)
Find a matching interaction based on the matchers and request.

This uses all of the matchers selected via configuration or `use_cassette` and passes in the request currently in progress.

Parameters **`request`** – `requests.PreparedRequest`

Returns *Interaction*

`is_empty`()
Determine if the cassette was empty when loaded.

`is_recording`()
Return whether the cassette is recording.

class `betamax.cassette.`**`Interaction`**(*interaction*, *response=None*)
The Interaction object represents the entirety of a single interaction.

The interaction includes the date it was recorded, its JSON representation, and the `requests.Response` object complete with its `request` attribute.

This object also handles the filtering of sensitive data.

No methods or attributes on this object are considered public or part of the public API. As such they are entirely considered implementation details and subject to change. Using or relying on them is not wise or advised.

`as_response`()
Return the Interaction as a Response object.

`deserialize`()
Turn a serialized interaction into a Response.

ignore()
Ignore this interaction.

This is only to be used from a before_record or a before_playback callback.

match(*matchers*)
Return whether this interaction is a match.

replace(*text_to_replace*, *placeholder*)
Replace sensitive data in this interaction.

replace_all(*replacements*, *serializing*)
Easy way to accept all placeholders registered.

Matchers

You can specify how you would like Betamax to match requests you are making with the recorded requests. You have the following options for default (built-in) matchers:

Matcher	Behaviour
body	This matches by checking the equality of the request bodies.
headers	This matches by checking the equality of all of the request headers
host	This matches based on the host of the URI
method	This matches based on the method, e.g., `GET`, `POST`, etc.
path	This matches on the path of the URI
query	This matches on the query part of the URI
uri	This matches on the entirety of the URI

Default Matchers

By default, Betamax matches on `uri` and `method`.

Specifying Matchers

You can specify the matchers to be used in the entire library by configuring Betamax like so:

```
import betamax

with betamax.Betamax.configure() as config:
    config.default_cassette_options['match_requests_on'].extend([
        'headers', 'body'
    ])
```

Instead of configuring global state, though, you can set it per cassette. For example:

```
import betamax
import requests


session = requests.Session()
recorder = betamax.Betamax(session)
match_on = ['uri', 'method', 'headers', 'body']
with recorder.use_cassette('example', match_requests_on=match_on):
    # ...
```

Making Your Own Matcher

So long as you are matching requests, you can define your own way of matching. Each request matcher has to inherit from `betamax.BaseMatcher` and implement `match`.

class `betamax.`**BaseMatcher**

Base class that ensures sub-classes that implement custom matchers can be registered and have the only method that is required.

Usage:

```
from betamax import Betamax, BaseMatcher

class MyMatcher(BaseMatcher):
    name = 'my'

    def match(self, request, recorded_request):
        # My fancy matching algorithm

Betamax.register_request_matcher(MyMatcher)
```

The last line is absolutely necessary.

The *match* method will be given a *requests.PreparedRequest* object and a dictionary. The dictionary always has the following keys:

- url
- method
- body
- headers

match(*request*, *recorded_request*)

A method that must be implemented by the user.

Parameters

- **request** (*PreparedRequest*) – A requests PreparedRequest object
- **recorded_request** (*dict*) – A dictionary containing the serialized request in the cassette

Returns bool True if they match else False

on_init()

Method to implement if you wish something to happen in `__init__`.

The return value is not checked and this is called at the end of `__init__`. It is meant to provide the matcher author a way to perform things during initialization of the instance that would otherwise require them to override `BaseMatcher.__init__`.

Some examples of matchers are in the source reproduced here:

```
# -*- coding: utf-8 -*-
from .base import BaseMatcher


class HeadersMatcher(BaseMatcher):
    # Matches based on the headers of the request
    name = 'headers'
```

```
    def match(self, request, recorded_request):
        return dict(request.headers) == self.flatten_headers(recorded_request)

    def flatten_headers(self, request):
        from betamax.util import from_list
        headers = request['headers'].items()
        return dict((k, from_list(v)) for (k, v) in headers)
```

```
# -*- coding: utf-8 -*-
from .base import BaseMatcher
from requests.compat import urlparse


class HostMatcher(BaseMatcher):
    # Matches based on the host of the request
    name = 'host'

    def match(self, request, recorded_request):
        request_host = urlparse(request.url).netloc
        recorded_host = urlparse(recorded_request['uri']).netloc
        return request_host == recorded_host
```

```
# -*- coding: utf-8 -*-
from .base import BaseMatcher


class MethodMatcher(BaseMatcher):
    # Matches based on the method of the request
    name = 'method'

    def match(self, request, recorded_request):
        return request.method == recorded_request['method']
```

```
# -*- coding: utf-8 -*-
from .base import BaseMatcher
from requests.compat import urlparse


class PathMatcher(BaseMatcher):
    # Matches based on the path of the request
    name = 'path'

    def match(self, request, recorded_request):
        request_path = urlparse(request.url).path
        recorded_path = urlparse(recorded_request['uri']).path
        return request_path == recorded_path
```

```
# -*- coding: utf-8 -*-
from .base import BaseMatcher
from requests.compat import urlparse


class PathMatcher(BaseMatcher):
    # Matches based on the path of the request
    name = 'path'
```

```
    def match(self, request, recorded_request):
        request_path = urlparse(request.url).path
        recorded_path = urlparse(recorded_request['uri']).path
        return request_path == recorded_path
```

```
# -*- coding: utf-8 -*-
from .base import BaseMatcher
from .query import QueryMatcher
from requests.compat import urlparse


class URIMatcher(BaseMatcher):
    # Matches based on the uri of the request
    name = 'uri'

    def on_init(self):
        # Get something we can use to match query strings with
        self.query_matcher = QueryMatcher().match

    def match(self, request, recorded_request):
        queries_match = self.query_matcher(request, recorded_request)
        request_url, recorded_url = request.url, recorded_request['uri']
        return self.all_equal(request_url, recorded_url) and queries_match

    def parse(self, uri):
        parsed = urlparse(uri)
        return {
            'scheme': parsed.scheme,
            'netloc': parsed.netloc,
            'path': parsed.path,
            'fragment': parsed.fragment
            }

    def all_equal(self, new_uri, recorded_uri):
        new_parsed = self.parse(new_uri)
        recorded_parsed = self.parse(recorded_uri)
        return (new_parsed == recorded_parsed)
```

When you have finished writing your own matcher, you can instruct betamax to use it like so:

```
import betamax

class MyMatcher(betamax.BaseMatcher):
    name = 'my'

    def match(self, request, recorded_request):
        return True

betamax.Betamax.register_request_matcher(MyMatcher)
```

To use it, you simply use the name you set like you use the name of the default matchers, e.g.:

```
with Betamax(s).use_cassette('example', match_requests_on=['uri', 'my']):
    # ...
```

on_init

As you can see in the code for `URIMatcher`, we use `on_init` to initialize an attribute on the `URIMatcher` instance. This method serves to provide the matcher author with a different way of initializing the object outside of the `match` method. This also means that the author does not have to override the base class' `__init__` method.

Serializers

You can tell Betamax how you would like it to serialize the cassettes when saving them to a file. By default Betamax will serialize your cassettes to JSON. The only default serializer is the JSON serializer, but writing your own is very easy.

Creating Your Own Serializer

Betamax handles the structuring of the cassette and writing to a file, your serializer simply takes a dictionary and returns a string.

Every Serializer has to inherit from *betamax.BaseSerializer* and implement three methods:

- `betamax.BaseSerializer.generate_cassette_name` which is a static method. This will take the directory the user (you) wants to store the cassettes in and the name of the cassette and generate the file name.
- `betamax.BaseSerializer.seralize()` is a method that takes the dictionary and returns the dictionary serialized as a string
- *betamax.BaseSerializer.deserialize()* is a method that takes a string and returns the data serialized in it as a dictionary.

class `betamax.`**BaseSerializer**
Base Serializer class that provides an interface for other serializers.

Usage:

```
from betamax import Betamax, BaseSerializer


class MySerializer(BaseSerializer):
    name = 'my'

    @staticmethod
    def generate_cassette_name(cassette_library_dir, cassette_name):
        # Generate a string that will give the relative path of a
        # cassette

    def serialize(self, cassette_data):
        # Take a dictionary and convert it to whatever

    def deserialize(self, cassette_data):
        # Uses a cassette file to return a dictionary with the
        # cassette information

Betamax.register_serializer(MySerializer)
```

The last line is absolutely necessary.

deserialize(*cassette_data*)
A method that must be implemented by the Serializer author.

The return value is extremely important. If it is not empty, the dictionary returned must have the following structure:

```
{
    'http_interactions': [{
        # Interaction
    },
    {
        # Interaction
    }],
    'recorded_with': 'name of recorder'
}
```

Params str cassette_data The data serialized as a string which needs to be deserialized.

Returns dictionary

on_init()

Method to implement if you wish something to happen in `__init__`.

The return value is not checked and this is called at the end of `__init__`. It is meant to provide the matcher author a way to perform things during initialization of the instance that would otherwise require them to override `BaseSerializer.__init__`.

serialize(*cassette_data*)

A method that must be implemented by the Serializer author.

Parameters **cassette_data** (*dict*) – A dictionary with two keys: `http_interactions`, `recorded_with`.

Returns Serialized data as a string.

Here's the default (JSON) serializer as an example:

```
from .base import BaseSerializer

import json
import os


class JSONSerializer(BaseSerializer):
    # Serializes and deserializes a cassette to JSON
    name = 'json'

    @staticmethod
    def generate_cassette_name(cassette_library_dir, cassette_name):
        return os.path.join(cassette_library_dir,
                            '{0}.{1}'.format(cassette_name, 'json'))

    def serialize(self, cassette_data):
        return json.dumps(cassette_data)

    def deserialize(self, cassette_data):
        try:
            deserialized_data = json.loads(cassette_data)
        except ValueError:
            deserialized_data = {}

        return deserialized_data
```

This is incredibly simple. We take advantage of the `os.path` to properly join the directory name and the file name. Betamax uses this method to find an existing cassette or create a new one.

Next we have the `betamax.serializers.JSONSerializer.serialize()` which takes the cassette dictionary and turns it into a string for us. Here we are just leveraging the `json` module and its ability to dump any valid dictionary to a string.

Finally, there is the `betamax.serializers.JSONSerializer.deserialize()` method which takes a string and turns it into the dictionary that betamax needs to function.

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

T

U