bravado

Nov 04, 2019

Contents

Bravado is a python client library for Swagger 2.0 services.

More information on Swagger can be found on the Swagger website

It aims to be a complete replacement to swagger codegen.

Features include:

- Dynamically generated client - no code generation needed!
- Synchronous and Asynchronous http clients out of the box.
- Strict validations to verify that your Swagger Schema is v2.0 compatible.
- HTTP request and response validation against your Swagger Schema.
- Swagger models as Python types (no need to deal with JSON).
- REPL friendly navigation of your Swagger schema with docstrings for Resources, Operations and Models.
- Ingestion of your Swagger schema via http or a local file path.

Contents:

CHAPTER 1

Quickstart

1.1 Usage

Install the latest stable version from PyPi:

```
$ pip install --upgrade bravado
```

1.2 Your first Hello World! (or Hello Pet)

Here is a simple example to try from a REPL (like IPython):

```
from bravado.client import SwaggerClient

client = SwaggerClient.from_url("http://petstore.swagger.io/v2/swagger.json")
pet = client.pet.getPetById(petId=42).response().result
```

If you were lucky, and pet Id with 42 was present, you will get back a result. It will be a dynamically created instance of `bravado.model.Pet` with attributes `category`, etc. You can even try `pet.category.id` or `pet.tags[0]`.

Sample Response:

```
Pet(category=Category(id=0L, name=u''), status=u'', name=u'', tags=[Tag(id=0L, name=u'
↪')], photoUrls=[u''], id=2)
```

If you got a `404`, try some other petId.

1.3 Lets try a POST call

Here we will demonstrate how `bravado` hides all the `JSON` handling from the user, and makes the code more Pythonic.

```
Pet = client.get_model('Pet')
Category = client.get_model('Category')
pet = Pet(id=42, name="tommy", category=Category(id=24))
client.pet.addPet(body=pet).response().result
```

1.4 Time to get Twisted! (Asynchronous client)

`bravado` provides an out of the box asynchronous http client with an optional timeout parameter.

Your first Hello World! (or Hello Pet) above can be rewritten to use the asynchronous Fido client like so:

```
from bravado.client import SwaggerClient
from bravado.fido_client import FidoClient

client = SwaggerClient.from_url(
    'http://petstore.swagger.io/v2/swagger.json',
    FidoClient()
)

result = client.pet.getPetById(petId=42).result(timeout=4)
```

Note: `timeout` parameter here is the timeout (in seconds) the call will block waiting for the complete response. The default timeout is to wait indefinitely.

Note: To use Fido client you should install bravado with fido extra via `pip install bravado[fido]`.

1.5 This is too fancy for me! I want a simple dict response!

`bravado` has taken care of that as well. Configure the client to not use models.

```
from bravado.client import SwaggerClient

client = SwaggerClient.from_url(
    'http://petstore.swagger.io/v2/swagger.json',
    config={'use_models': False}
)

result = client.pet.getPetById(petId=42).result(timeout=4)
```

`result` will look something like:

```
{
    'category': {
```

(continues on next page)

(continued from previous page)

```
        'id': 0L,
        'name': u''
    },
    'id': 2,
    'name': u'',
    'photoUrls': [u''],
    'status': u'',
    'tags': [
        {'id': 0L, 'name': u''}
    ]
}
```

CHAPTER 2

Configuration

2.1 Client Configuration

You can configure certain behaviours when creating a `SwaggerClient`.

bravado and bravado-core use the same config dict. The full documentation for bravado-core config keys is available too.

```
from bravado.client import SwaggerClient, SwaggerFormat

my_super_duper_format = SwaggerFormat(...)

config = {
    # === bravado config ===

    # What class to use for response metadata
    'response_metadata_class': 'bravado.response.BravadoResponseMetadata',

    # Do not use fallback results even if they're provided
    'disable_fallback_results': False,

    # DEPRECATED: Determines what is returned by HttpFuture.result().
    # Please use HttpFuture.response() for accessing the http response.
    'also_return_response': False,

    # === bravado-core config ====

    # Validate incoming responses
    'validate_responses': True,

    # Validate outgoing requests
    'validate_requests': True,

    # Validate the swagger spec
```

(continues on next page)

(continued from previous page)

```
    'validate_swagger_spec': True,

    # Use models (Python classes) instead of dicts for #/definitions/{models}
    'use_models': True,

    # List of user-defined formats
    'formats': [my_super_duper_format],

}

client = SwaggerClient.from_url(..., config=config)
```

Config key	Type	Description
response_metadata_class	string	The Metadata class to use; see *Custom response metadata* for details. Default: `bravado.response.BravadoResponseMetadata`
disable_fallback_results	boolean	Whether to disable returning fallback results, even if they're provided as an argument to to `HttpFuture.response()`. Default: `False`
also_return_response	boolean	Determines what is returned by the service call. Specifically, the return value of `HttpFuture.result()`. When `False`, the swagger result is returned. When `True`, the tuple `(swagger result, http response)` is returned. Has no effect on the return value of `HttpFuture.response()`. See *Getting access to the HTTP response*. Default: `False`

2.1.1 Customizing the HTTP client

bravado's default HTTP client uses the excellent requests library to make HTTP calls. If you'd like to customize its behavior, create a `bravado.requests_client.RequestsClient` instance yourself and pass it as `http_client` argument to `SwaggerClient.from_url()` or `SwaggerClient.from_spec()`.

Currently, you can customize SSL/TLS behavior through the arguments `ssl_verify` and `ssl_cert`. They're identical to the `verify` and `cert` options of the requests library; please check their documentation for usage instructions. Note that bravado honors the `REQUESTS_CA_BUNDLE` environment variable as well.

Also you can specify custom future adapter and response adapter classes through the `future_adapter_class` and `response_adapter_class` arguments respectively.

2.1.2 Using a different HTTP client

You can use other HTTP clients with bravado; the fido client ships with bravado (`bravado.fido_client.FidoClient`). Currently the fido client doesn't support customizing SSL/TLS behavior. But the future adapter and response adapter classes could be specified in the same manner as for `bravado.requests_client.RequestsClient` - through the `future_adapter_class` and `response_adapter_class` arguments respectively.

Another well-supported option is bravado_asyncio, which requires Python 3.5+. It supports the same ssl options as the default requests client.

2.2 Per-request Configuration

Configuration can also be applied on a per-request basis by passing in `_request_options` to the service call.

```
client = SwaggerClient.from_url(...)
request_options = { ... }
client.pet.getPetById(petId=42, _request_options=request_options).response().result
```

Config key	Type	Default	Description
connect_timeout	float	N/A	TCP connect timeout in seconds. This is passed along to the http_client when making a service call.
headers	dict	N/A	Dict of http headers to to send with the outgoing request.
response_callbacks	list of callables	[]	List of callables that are invoked after the incoming response has been validated and unmarshalled but before being returned to the calling client. This is useful for client decorators that would like to hook into the post-receive event. The callables are executed in the order they appear in the list. Two parameters are passed to each callable: - `incoming_response` of type `bravado_core.response.IncomingResponse` - `operation` of type `bravado_core.operation.Operation`
timeout	float	N/A	TCP idle timeout in seconds. This is passed along to the http_client when making a service call.
use_msgpack	boolean	False	If a msgpack serialization is desired for the response. This

CHAPTER 3

Making requests with bravado

When you call `SwaggerClient.from_url()` or `SwaggerClient.from_spec()`, Bravado takes a Swagger (OpenAPI) 2.0 spec and returns a `SwaggerClient` instance that you can use to make calls to the service described in the spec. You make calls by doing Python method calls in the form of `client.resource.operation(operation_params)`. Use `dir(client)` to see all available resources.

3.1 Resources and operations

Resources are generated for each tag that exists in your Swagger spec. If an operation has no tags then the left-most element of its path is taken as resource name. So in the case of an operation with the path `/pet/find`, `pet` will be the resource.

The operation name will be the (*sanitized*) operationId value from the Swagger spec. If there is no operationId, it will be generated. We highly recommend providing operation IDs for all operations. Use `dir(client.resource)` to see a list of all available operations.

The operation method expects keyword arguments that have the same (*sanitized*) names as in the Swagger spec. Use corresponding Python types for the values - if the Swagger spec says a parameter is of type `boolean`, provide it as a Python `bool`.

3.2 Futures and responses

The return value of the operation method is a `HttpFuture`. To access the response, call `HttpFuture.response()`. This call will block, i.e. it will wait until the response is received or the timeout you specified is reached.

If the request succeeded and the server returned a HTTP status code between 100 and 299, the return value of `HttpFuture.response()` will be a `BravadoResponse` instance. You may access the Swagger result of your call through `BravadoResponse.result`.

If the server sent a response with a HTTP code of 300 or higher, by default a subclass of `HTTPError` will be raised when you call `HttpFuture.response()`. The exception gives you access to the Swagger result (`HTTPError.swagger_result`) as well as the HTTP response object (`HTTPError.response`).

3.3 Response metadata

`BravadoResponse.metadata` is an instance of `BravadoResponseMetadata` that provides you with access to the HTTP response including headers and HTTP status code, request timings and whether a fallback result was used (see *Working with fallback results*).

You're able to provide your own implementation of `BravadoResponseMetadata`; see *Custom response metadata* for details.

3.4 Sanitizing names

Not all characters that the Swagger spec allows for names are valid Python identifiers. In particular, spaces and the `-` character can be troublesome. bravado sanitizes resource, operation and parameter names according to these rules:

- Any character that is not a letter or number is converted to an underscore (_)
- Collapse multiple consecutive underscores to one
- Remove leading and trailing underscores
- Remove leading numbers

CHAPTER 4

Advanced Usage

4.1 Validations

`bravado` validates the schema against the Swagger 2.0 Spec. Validations are also done on the requests and the responses.

Validation example:

```
pet = Pet(id="I should be integer :(", name="tommy")
client.pet.addPet(body=pet).response().result
```

will result in an error like so:

```
TypeError: id's value: 'I should be integer :(' should be in types (<type 'long'>,
↪<type 'int'>)
```

Note: If you'd like to disable validation of outgoing requests, you can set `validate_requests` to `False` in the `config` passed to `SwaggerClient.from_url(...)`.

The same holds true for incoming responses with the `validate_responses` config option.

4.2 Adding Request Headers

`bravado` allows you to pass request headers along with any request.

```
Pet = client.get_model('Pet')
Category = client.get_model('Category')
pet = Pet(id=42, name="tommy", category=Category(id=24))
swagger_client.pet.addPet(
    body=pet,
```

(continues on next page)

(continued from previous page)

```
    _request_options={"headers": {"foo": "bar"}},
).response().result
```

4.3 Docstrings

`bravado` provides docstrings to operations and models to quickly get the parameter and response types. Due to an implementation limitation, an operation's docstring looks like a class docstring instead of a function docstring. However, the most useful information about parameters and return type is present in the `Docstring` section.

Note: The `help` built-in does not work as expected for docstrings. Use the `?` method instead.

```
>> petstore.pet.getPetById?

Type:       CallableOperation
String Form:<bravado.client.CallableOperation object at 0x241b5d0>
File:       /some/dir/bravado/bravado/client.py
Definition: c.pet.getPetById(self, **op_kwargs)
Docstring:
[GET] Find pet by ID

Returns a single pet

:param petId: ID of pet to return
:type petId: integer
:returns: 200: successful operation
:rtype: object
:returns: 400: Invalid ID supplied
:returns: 404: Pet not found
Constructor Docstring::type operation: :class:`bravado_core.operation.Operation`
Call def:   c.pet.getPetById(self, **op_kwargs)
Call docstring:
Invoke the actual HTTP request and return a future that encapsulates
the HTTP response.

:rtype: :class:`bravado.http_future.HTTPFuture`
```

Docstrings for models can be retrieved as expected:

```
>> pet_model = petstore.get_model('Pet')
>> pet_model?

Type:       type
String Form:<class 'bravado_core.model.Pet'>
File:       /some/dir/bravado_core/model.py
Docstring:
Attributes:

category: Category
id: integer
name: string
photoUrls: list of string
status: string - pet status in the store
```

(continues on next page)

(continued from previous page)

```
tags: list of Tag
Constructor information:
 Definition:pet_type(self, **kwargs)
```

4.4 Default Values

`bravado` uses the default values from the spec if the value is not provided in the request.

In the Pet Store example, operation `findPetsByStatus` has a `default` of `available`. That means, `bravado` will plug that value in if no value is provided for the parameter.

```
client.pet.findPetByStatus()
```

4.5 Loading swagger.json by file path

`bravado` also accepts `swagger.json` from a file path. Like so:

```
client = SwaggerClient.from_url('file:///some/path/swagger.json')
```

Alternatively, you can also use the `load_file` helper method.

```
from bravado.swagger_model import load_file

client = SwaggerClient.from_spec(load_file('/path/to/swagger.json'))
```

4.6 Getting access to the HTTP response

The default behavior for a service call is to return the swagger result like so:

```
pet = client.pet.getPetById(petId=42).response().result
print pet.name
```

However, there are times when it is necessary to have access to the actual HTTP response so that the HTTP headers or HTTP status code can be used. Simply save the response object (which is a `BravadoResponse`) and use its `incoming_response` attribute to access the incoming response:

```
petstore = SwaggerClient.from_url(
    'http://petstore.swagger.io/v2/swagger.json',
    config={'also_return_response': True},
)
pet_response = petstore.pet.getPetById(petId=42).response()
http_response = pet_response.incoming_response
assert isinstance(http_response, bravado_core.response.IncomingResponse)
print http_response.headers
print http_response.status_code
print pet.name
```

4.7 Working with fallback results

By default, if the server returns an error or doesn't respond in time, you have to catch and handle the resulting exception accordingly. A simpler way would be to use the support for fallback results provided by `HttpFuture.response()`.

`HttpFuture.response()` takes an optional argument `fallback_result` which is the fallback Swagger result to return in case of errors:

```
petstore = SwaggerClient.from_url('http://petstore.swagger.io/v2/swagger.json')
response = petstore.pet.findPetsByStatus(status=['available']).response(
    timeout=0.5,
    fallback_result=[],
)
```

This code will return an empty list in case the server doesn't respond quickly enough (or it responded quickly enough, but returned an error).

4.7.1 Handling error types differently

Sometimes, you might want to treat timeout errors differently from server errors. To do this you may pass in a callable as `fallback_result` argument. The callable takes one mandatory argument: the exception that would have been raised normally. This allows you to return different results based on the type of error (e.g. a `BravadoTimeoutError`) or, if a server response was received, on any data pertaining to that response, like the HTTP status code. Subclasses of `HTTPError` have a `response` attribute that provides access to that data.

```
def pet_status_fallback(exc):
    if isinstance(exc, BravadoTimeoutError):
        # Backend is slow, return last cached response
        return pet_status_cache

    # Some server issue, let's not show any pets
    return []

petstore = SwaggerClient.from_url(
    'http://petstore.swagger.io/v2/swagger.json',
    # The petstore result for this call is not spec compliant...
    config={'validate_responses': False},
)
response = petstore.pet.findPetsByStatus(status=['available']).response(
    timeout=0.5,
    fallback_result=pet_status_fallback,
)

if not response.metadata.is_fallback_result:
    pet_status_cache = response.result
```

4.7.2 Customizing which error types to handle

By default, the fallback result will be used either when the server doesn't send the response in time or when it returns a server error (i.e. a result with a HTTP 5XX status code). To override this behavior, specify the `exceptions_to_catch` argument to `HttpFuture.response()`.

The default is defined in `bravado.http_future.FALLBACK_EXCEPTIONS`. See `bravado.exception` for a list of possible exception types.

4.7.3 Models and fallback results

But what if you're using models (the default) and the endpoint you're calling returns one? You'll have to return one as well from your fallback_result function to stay compatible with the rest of your code:

```
petstore = SwaggerClient.from_url('http://petstore.swagger.io/v2/swagger.json')
response = petstore.pet.getPetById(petId=101).response(
    timeout=0.5,
    fallback_result=petstore.get_model('Pet')(name='No Pet found', photoUrls=[]),
)
```

Two things to note here: first, use `SwaggerClient.get_model()` to get the model class for a model name. Second, since `name` and `photoUrls` are required fields for this model, we probably should not leave them empty (if we do they'd still be accessible, but the value would be `None`). It's up to you how you decide to deal with this case.

`BravadoResponseMetadata.is_fallback_result` will be True if a fallback result has been returned by the call to `HttpFuture.response()`.

4.7.4 Testing fallback results

You can trigger returning fallback results for testing purposes. Just set the option `force_fallback_result` to `True` in the request configuration (see *Per-request Configuration*). In this case a `ForcedFallbackResultError` exception will be passed to your fallback result callback, so make sure you handle it properly.

4.8 Custom response metadata

Sometimes, there's additional metadata in the response that you'd like to make available easily. This case arises most often if you're using bravado to talk to internal services. Maybe you have special HTTP headers that indicate whether a circuit breaker was triggered? bravado allows you to customize the metadata and provide custom attributes and methods.

In your code, create a class that subclasses `bravado.response.BravadoResponseMetadata`. In the implementation of your properties, use `BravadoResponseMetadata.headers` to access response headers, or `BravadoResponseMetadata.incoming_response` to access any other part of the HTTP response.

If, for some reason, you need your own `__init__` method, make sure that your signature accepts any positional and keyword argument, and that you call the base method with these arguments from your own implementation. That way, your class will remain compatible with the base class even if new arguments get added to the __init__ method. Example minimal implementation:

```
class MyResponseMetadata(ResponseMetadata):
    def __init__(self, *args, **kwargs):
        super(MyResponseMetadata, self).__init__(*args, **kwargs)
```

While developing custom `BravadoResponseMetadata` classes we recommend to avoid, if possible, the usage of attributes for data that's expensive to compute. Since the object will be created for every response, implementing these fields as properties makes sure the evaluation is only done if the field is accessed.

CHAPTER 5

Testing code that uses bravado

Writing tests is crucial in making sure your code works and behaves as expected. bravado ships with two classes that will help you create robust unit tests to verify the correctness of your code. We'll be using the excellent pytest library in this example.

First of all, let's define the code we'd like to test:

```
from itertools import chain

from bravado.client import SwaggerClient

def get_available_pet_photos():
    petstore = SwaggerClient.from_url(
        'http://petstore.swagger.io/v2/swagger.json',
    )
    pets = petstore.pet.findPetsByStatus(status=['available']).response(
        timeout=0.5,
        fallback_result=lambda e: [],
    ).result

    return chain.from_iterable(pet.photoUrls for pet in pets)
```

First of all, in order to make sure your code doesn't do any network requests, you need to mock out the bravado client:

```
import mock
import pytest

from bravado.client import SwaggerClient

@pytest.fixture
def mock_client():
    mock_client = mock.Mock(name='mock SwaggerClient')
    with mock.patch.object(SwaggerClient, 'from_url', return_value=mock_client):
        yield mock_client
```

Now we can mock out that call to `findPetsByStatus` by using the `bravado.testing.response_mocks.BravadoResponseMock` class:

```
import mock

from bravado.testing.response_mocks import BravadoResponseMock

from mypackage import get_available_pet_photos

def test_get_available_pet_photos(mock_client):
    mock_pets = [
        mock.Mock(
            photoUrls=['https://example.com/image.png'],
        ),
        mock.Mock(
            photoUrls=[
                'https://example.com/image2.png',
                'https://example.com/image3.png',
            ],
        ),
    ]

    mock_client.pet.findPetsByStatus.return_value.response = BravadoResponseMock(
        result=mock_pets,
    )

    pet_photos = get_available_pet_photos()

    assert list(pet_photos) == [
        'https://example.com/image.png',
        'https://example.com/image2.png',
        'https://example.com/image3.png',
    ]
```

Note that it's your responsibility to ensure that what you set as result for `BravadoResponseMock` is sufficiently similar to what bravado would return in production. We've used a `Mock` class here; another option is to define namedtuples that correspond to your Swagger spec objects. This gives you even greater confidence in the correctness of your code since access to undefined fields will result in an error.

5.1 Testing degraded responses

Use `FallbackResultBravadoResponseMock` to test *fallback results*. It works similarly, but you don't have to pass the result to the constructor, since your fallback result callback will determine the result. Let's add another test to verify our fallback result code path works properly:

```
from bravado.testing.response_mocks import FallbackResultBravadoResponseMock

from example import get_available_pet_photos

def test_get_available_pet_photos_fallback_result(mock_client):
    mock_client.pet.findPetsByStatus.return_value\
        .response = FallbackResultBravadoResponseMock()

    pet_photos = get_available_pet_photos()

    assert list(pet_photos) == []
```

Note that you can pass in a custom exception instance to `FallbackResultBravadoResponseMock` if you need to trigger specific exception handling in your fallback result callback.

5.2 Setting custom response metadata

Both `BravadoResponseMock` as well as `FallbackResultBravadoResponseMock` accept an optional `metadata` argument. Just pass in an instance of `BravadoResponseMetadata` that you'd like to be used. A default one will be provided otherwise.

CHAPTER 6

API reference

6.1 bravado Package

6.1.1 `bravado` Package

6.1.2 `client` Module

6.1.3 `config` Module

6.1.4 `requests_client` Module

6.1.5 `fido_client` Module

6.1.6 `http_future` Module

6.1.7 `response` Module

6.1.8 `exception` Module

6.1.9 `testing` Module

CHAPTER 7

Changelog

7.1 10.4.3 (2019-11-04)

- Some type annotation fixes and improvements, in particular for the response_adapter argument to `HttpFuture` - PR #436

7.2 10.4.2 (2019-10-16)

- Show the response text when an unexpected 5xx response is returned. This fixes a regression introduced in PR #32 - PR #425. Thanks Kevin Coleman for your contribution!
- Allow custom future and response adapter classes in both `RequestsClient` and `FidoClient` - PR #430, #431. Thanks Andrii Stepaniuk for your contribution!
- Fix `HttpErrorType` type annotations - PR #433

7.3 10.4.1 (2019-05-08)

- Fix and re-enable integration tests on Windows - PR #417

7.4 10.4.0 (2019-05-07)

- Ensure that *bravado* supports Windows platform - PR #415
- Ensure that responses with *application/.** content type returns raw binary data - PR # 414. Thanks Greg Ruane for your contribution!

7.5 10.3.2 (2019-03-25)

- Do not warn about conflicting timeouts in requests HTTP client if only one was specified - PR #411. Thanks Pokey Rule for your contribution!
- Fix bug in `bravado.fido_client.FidoResponseAdapter`, it was not returning a unicode string for `text` - PR #412

7.6 10.3.1 (2019-02-27)

- Fix `bravado.response.BravadoResponseMetadata.is_fallback_result`, it was always `True` in 10.3.0 - Issue #409, PR #410
- `bravado.response.BravadoResponseMetadata.handled_exception_info` is `None` again if no exception was handled - PR #410
- `bravado.testing.response_mocks` is now type-annotated - PR #410

7.7 10.3.0 (2019-02-20)

- bravado is now fully type-annotated - PR #403
- Add ability to cancel a HttpFuture. Third-party HTTP clients will need to implement `cancel` on their `bravado.http_future.FutureAdapter` class to support this - PR #406
- The static method `from_config_dict` of `bravado.config.BravadoConfig` was removed due to compatibility issues with Python 3.5.0. This method was meant for internal use only; if you do happen to call it please switch to `bravado.config.bravado_config_from_config_dict()` instead. - PR #407

7.8 10.2.2 (2019-01-03)

- Fix issue with default (requests) HTTP client if HTTP_PROXY environment variable is set - Issue #401, PR #402. Thanks Lourens Veen for the initial report!

7.9 10.2.1 (2018-11-16)

- Reraise network errors when unmarshalling - PR #397

7.10 10.2.0 (2018-10-19)

- Support customizing or disabling SSL/TLS validation for the default HTTP client - Issues #278, #311, PR #392
- Use the fallback result in case of connection errors as well - PR #381

7.11 10.1.0 (2018-06-26)

- Add support for non-callable fallback results, stabilize the response API - PR #376
- Add unified connection error handling support, introduce `bravado.exception.BravadoConnectionError` - PR #377
- Support per-request API key header overwriting - PR #374. Thanks Yuliya Bagriy for your contribution!
- Extract integration testing tools to `bravado.testing.integration_test` module - PR #378

7.12 10.0.1 (2018-06-20)

- Add helper classes (in `bravado.testing.response_mocks`) for unit testing code using bravado - PR #375

7.13 10.0.0 (2018-06-15)

- Re-add ability to force returning fallback results - PR #372. Per-request configuration is now handled by the new `bravado.config.RequestConfig` class. This change requires an updated version of bravado-asyncio in case you're using that HTTP client.

7.14 9.3.2 (2018-06-15)

- Revert ability to force returning fallback results which was introduced in 9.3.1, since it contains backwards-incompatible changes that break third-party HTTP clients like bravado-asyncio.

7.15 9.3.1 (2018-06-14)

- Add ability to force returning fallback results - PR #372

7.16 9.3.0 (2018-06-05)

- Introduce the HTTPFuture.response API as well as support for returning a fallback result. - PR #365, #366, #367, #368

  NOTE: Most of this API is not documented yet and is considered experimental; we're working on stabilizing it and providing developer documentation.

7.17 9.2.2 (2017-12-19)

- Fix msgpack import issue - PR #341. Thanks Jesse Myers for your contribution!

7.18 9.2.1 (2017-12-07)

- The timeout exception for the requests client should inherit from `requests.exceptions.ReadTimeout` instead of `requests.exceptions.Timeout` - PR #337

7.19 9.2.0 (2017-11-10)

- Support msgpack as wire format for response data - PR #323, 328, 330, 331
- Allow client to access resources for tags which are not valid Python identifier names, by adding the `SwaggerClient.get_resource` method. For example, `client.get_resource('My Pets').list_pets()` - PR #320. Thanks Craig Blaszczyk for your contribution!
- Unify timeout exception classes. You can now simply catch `bravado.exception.BravadoTimeoutError` (or `builtins.TimeoutError` if you're using Python 3.3+) - PR #321

7.20 9.1.1 (2017-10-10)

- Allow users to pass the tcp_nodelay request parameter to FidoClient requests - PR #319

7.21 9.1.0 (2017-08-02)

- Make sure HTTP header names and values are unicode strings when using the fido HTTP client. NOTE: this is a potentially backwards incompatible change if you're using the fido HTTP client and are working with response headers. It's also highly advised to not upgrade to bravado-core 4.8.0+ if you're using fido unless you're also upgrading to a bravado version that contains this change.

7.22 9.0.7 (2017-07-05)

- Require fido version 4.2.1 so we stay compatible to code catching crochet.TimeoutError

7.23 9.0.6 (2017-06-28)

- Don't mangle headers with bytestring values on Python 3

7.24 9.0.5 (2017-06-23)

- Make sure headers passed in for fetching specs are converted to str as well

7.25 9.0.4 (2017-06-22)

- Fix regression when passing swagger parameters of type header in `_request_options` introduced by PR #288

7.26 9.0.3 (2017-06-21)

- When using the fido HTTP client and passing a timeout to `result()`, make sure we throw a fido HTTPTimeoutError instead of a crochet TimeoutError when hitting the timeout.

7.27 9.0.2 (2017-06-12)

- `_requests_options` headers are casted to `string` to support newer version of `requests` library.

7.28 9.0.1 (2017-06-09)

- Convert http method to str while constructing the request to fix an issue with file uploads when using requests library versions before 2.8.

7.29 9.0.0 (2017-06-06)

- Add API key authentication via header to RequestsClient.
- Fido client is now an optional dependency. **NOTE**: if you intend to use bravado with the fido client you need to install bravado with fido extras (`pip install bravado[fido]`)

7.30 8.4.0 (2016-09-27)

- Remove support for Python 2.6, fixing a build failure.
- Switch from Python 3.4 to Python 3.5 for tests.

7.31 8.3.0 (2016-06-03)

- Bravado using Fido 3.2.0 python 3 ready

7.32 8.2.0 (2016-04-29)

- Bravado compliant to Fido 3.0.0
- Dropped use of concurrent futures in favor of crochet EventualResult
- Workaround for bypassing a unicode bug in python *requests* < 2.8.1

7.33 8.1.2 (2016-04-18)

- Don’t unnecessarily constrain the version of twisted when not using python 2.6

7.34 8.1.1 (2016-04-13)

- Removed logic to build multipart forms. Using python 'requests' instead to build the entire http request.

7.35 8.1.0 (2016-04-04)

- Support for YAML Swagger specs - PR #198
- Remove pytest-mock dependency from requirements-dev.txt. No longer used and it was breaking the build.
- Requires bravado-core >= 4.2.2
- Fix unit test for default values getting sent in the request

7.36 8.0.1 (2015-12-02)

- Require twisted < 15.5.0 since Python 2.6 support was dropped

7.37 8.0.0 (2015-11-25)

- Support for recursive $refs
- Support for remote $refs e.g. Swagger 2.0 specs that span multiple json files
- Requires bravado-core 4.0.0 which is not backwards compatible (See its CHANGELOG)
- Transitively requires swagger-spec-validator 2.0.2 which is not backwards compatible (See its CHANGELOG)

7.38 7.0.0 (2015-10-23)

- Support per-request response_callbacks to enable `SwaggerClient` decorators to instrument an `IncomingResponse` post-receive. This is a non-backwards compatible change iff you have implemented a custom `HttpClient`. Consult the changes in signature to `HttpClient.request()` and `HttpFuture`'s constructor.
- Config option `also_return_response` is supported on a per-request basis.

7.39 6.1.1 (2015-10-19)

- Fix `IncomingResponse` subclasses to provide access to the http headers.
- Requires bravado-core >= 3.1.0

7.40 6.1.0 (2015-10-19)

- Clients can now access the HTTP response from a service call to access things like headers and status code. See Advanced Usage

7.41 6.0.0 (2015-10-12)

- User-defined formats are no longer global. The registration mechanism has changed and is now done via configuration. See Configuration

7.42 5.0.0 (2015-08-27)

- Update ResourceDecorator to return an operation as a CallableOperation instead of a function wrapper (for the docstring). This allows further decoration of the ResourceDecorator.

7.43 4.0.0 (2015-08-10)

- Consistent bravado.exception.HTTPError now thrown from both Fido and Requests http clients.
- HTTPError refactored to contain an optional detailed message and Swagger response result.

7.44 3.0.0 (2015-08-03)

- Support passing in connect_timeout and timeout via _request_options to the Fido and Requests clients
- Timeout in HTTPFuture now defaults to None (wait indefinitely) instead of 5s. You should make sure any calls to http_future.result(..) without a timeout are updated accordingly.

7.45 2.1.0 (2015-07-20)

- Add warning for deprecated operations

7.46 2.0.0 (2015-07-13)

- Assume responsibility for http invocation (used to be in bravado-core)

7.47 1.1.0 (2015-07-06)

- Made bravado compatible with Py34

7.48 1.0.0 (2015-06-26)

- Fixed petstore demo link
- Pick up bug fixes from bravado-core 1.1.0

7.49 1.0.0-rc2 (2015-06-01)

- Renamed ResponseLike to IncomingResponse to match bravado-core

7.50 1.0.0-rc1 (2015-05-13)

- Initial version - large refactoring/rewrite of swagger-py 0.7.5 to support Swagger 2.0

CHAPTER 8

Indices and tables

- genindex
- modindex