bravado$_{core}$

Release 4.8.4

Jun 14, 2018

Contents

bravado_core is a Python library that implements the Swagger 2.0 Specification.

Client and servers alike can use bravado_core to implement these features:

- Swagger Schema ingestion and validation
- Validation and marshalling of requests and responses
- Validation and marshalling of user-defined Swagger formats
- Modelling Swagger *#/definitions* as Python classes or dicts

For example:

- bravado uses bravado-core to implement a fully functional Swagger client.
- pyramid_swagger uses bravado-core to seamlessly add Swagger support to Pyramid webapps.

Contents:

CHAPTER 1

Configuration

All configuration is stored in a `dict`.

```
from bravado_core.spec import Spec

spec_dict = json.loads(open('swagger.json', 'r').read())

config = {
    'validate_requests': False,
    'use_models': False,
}

swagger_spec = Spec.from_dict(spec_dict, config=config)
```

Config key	Type	Default	Description
validate_swagger_spec	boolean	True	Validate the Swagger spec against the Swagger 2.0 Specification.
validate_requests	boolean	True	On the client side, validates outgoing requests. On the server side, validates incoming requests.
validate_responses	boolean	True	On the client side, validates incoming responses. On the server side, validates outgoing responses.
use_models	boolean	True	Use python classes to represent models instead of dicts. See *Python Models*.
formats	list of SwaggerFormat	[]	List of user-defined formats to support. See *User-Defined Formats*.
include_missing_properties	boolean	True	Create properties with the value `None` if they were not submitted during object unmarshalling

CHAPTER 2

Python Models

Models in a Swagger spec are usually defined under the path `#/definitions`.

A model can refer to a primitive type or a container type such as a `list` or a `dict`. In `dict` form, there is an opportunity to make the interface to access the properties of a model a little more straight forward.

Consider the following:

```
{
    "definitions": {
        "Pet": {
            "type": "object",
            "required": ["name"],
            "properties": {
                "name": {"type": "string"},
                "age": {"type": "integer"},
                "breed": {"type": "string"}
            }
        }
    }
}
```

In python, this model easily maps to a `dict`:

```
pet = {
    "name": "Sumi",
    "age": 12,
    "breed": None,
}

print pet['name']

if pet['age'] < 1:
    print 'What a cute puppy!'
```

(continues on next page)

(continued from previous page)

```
if pet['breed'] is None:
    pet['breed'] = 'mutt'
```

However, if the model is implemented as a Python type, dotted access to properties becomes a reality:

```
from bravado_core.spec import Spec

spec = Spec.from_dict(...)
Pet = spec.definitions['Pet']
pet = Pet(name='Sumi', age=12)

print pet.name

if pet.age < 1:
    print 'What a cute puppy!'

if pet.breed is None:
    pet.breed = 'mutt'
```

2.1 Configuring Models as Python Types

bravado-core supports models as both dicts and python types.

The feature to use python types for models is enabled by default. You can always disable it if necessary.

```
from bravado_core.spec import Spec
swagger_dict = {..}
spec = Spec.from_dict(swagger_dict, config={'use_models': False})
```

2.2 Allowing null values for properties

Typically, bravado-core will complain during validation if it encounters fields with `null` values. This can be problematic, especially when you're adding Swagger support to pre-existing APIs. In that case, declare your model properties as `x-nullable`:

```
{
    "Pet": {
        "type": "object",
        "properties": {
            "breed": {
                "type": "string",
                "x-nullable": true
            }
        }
    }
}
```

`x-nullable` is an extension to the Swagger 2.0 spec. A `nullable` attribute is being considered for the next major version of Swagger.

2.3 Model Discovery

Keep in mind that bravado-core has to do some extra legwork to figure out which parts of your spec represent Swagger models and which parts don't to make this feature work automagically. With a single-file Swagger spec, this is pretty straight forward - everything under `#/definitions` is a model. However, with more complicated specs that span multiple files and use external refs, it becomes a bit more involved. For this reason, the discovery process for models is best effort with a fallback to explicit annotations as follows:

1. Search for refs that refer to `#/definitions` in local scope
2. Search for refs that refer to external definitions with pattern `<filename>#/definitions/<model name>`.

   swagger.json

```
{
     "paths": {
         "/pet": {
             "get": {
                 "responses": {
                     "200": {
                         "description": "A pet",
                         "schema": {
                             "$ref": "another_file.json#/definitions/Pet"
                         }
                     }
                 }
             }
         }
     }
 }
```

   another_file.json

```
{
    "definitions": {
        "Pet": {
            ...
        }
    }
}
```

3. Search for the `"x-model": "<model name>"` annotation to identify models that can't be found via method 1. or 2.

   swagger.json

```
{
    "paths": {
        "/pet": {
            "get": {
                "responses": {
                    "200": {
                        "description": "A pet",
                        "schema": {
                            "$ref": "https://my.company.com/definitions/models.
↪json#/models/Pet"
                        }
```

(continues on next page)

(continued from previous page)

```
                        }
                    }
                }
            }
        }
}
```

models.json (served up via `https://my.company.com/definitions/models.json`)

```
{
    "models": {
        "Pet": {
             "x-model": "Pet"
            ...
        }
    }
}
```

CHAPTER 3

User-Defined Formats

Primitive types in Swagger support an optional modifier property `format` as explained in detail in the Swagger Specification. With this feature, you can define your own domain specific formats and have validation and marshalling to/from python/json handled transparently.

3.1 Creating a user-defined format

This is best explained with a simple example. Let's create a user-defined format for CIDR notation.

In a Swagger spec, the schema-object for a CIDR would resemble:

```
{
    "type": "string",
    "format": "cidr",
    "description": "IPv4 CIDR"
}
```

In python, we'd like CIDRs to automatically be converted to a `CIDR` object that makes them easy to work with.

```
class CIDR(object):
    def __init__(self, cidr):
        """
        :param cidr: CIDR in string form.
        """
        self.cidr = cidr

    def overlaps(self, other_cidr):
        """Return true if other_cidr overlaps with this cidr"""
        ...

    def subnet_mask(self):
        """Return the subnet mask of this cidr"""
        ...
```

(continues on next page)

(continued from previous page)

```
    ...
```

We would also like CIDRs to be validated by bravado-core whenever they are part of a HTTP request or response.

Create a `bravado_core.formatter.SwaggerFormat` to define the CIDR format:

```
from bravado_core.formatter import SwaggerFormat

def validate_cidr(cidr_string):
    if '/' not in cidr_string:
        raise SwaggerValidationError('CIDR {0} is invalid'.format(cidr_string))

cidr_format = SwaggerFormat(
    # name of the format as used in the Swagger spec
    format='cidr',

    # Callable to convert a python CIDR object to a string
    to_wire=lambda cidr_object: cidr_object.cidr,

    # Callable to convert a string to a python CIDR object
    to_python=lambda cidr_string: CIDR(cidr_string),

    # Callable to validate the cidr in string form
    validate=validate_cidr
)
```

3.2 Configuring user-defined formats

Now that we have a `cidr_format`, just pass it to a `Spec` as part of the `config` parameter on `Spec` creation.

```
from bravado_core.spec import Spec

spec_dict = json.loads(open('swagger.json', 'r').read())
config = {
    'validate_responses': True,
    'validate_requests': True,
    'formats': [cidr_format],
}
spec = Spec.from_dict(spec_dict, config=config)
```

All validation and processing of HTTP requests and responses will now use the configured format where appropriate.

3.3 Putting it all together

A simple example of passing a CIDR object to a request and getting a list of CIDR objects back from the response.

```
{
    "paths": {
        "/get_overlapping_cidrs": {
            "get": {
                "parameters": [
```

(continues on next page)

(continued from previous page)

```
                    {
                        "name": "cidr",
                        "in": "query",
                        "type": "string",
                        "format": "cidr"
                    }
                ],
                "responses": {
                    "200": {
                        "description": "List of overlapping cidrs",
                        "schema": {
                            "type": "array",
                            "items": {
                                "type": "string",
                                "format": "cidr"
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```
from bravado_core.spec import Spec
from bravado_core.response import unmarshal_response
from bravado_core.param import marshal_param

# Retrieve the swagger spec from the server and json.load() it
spec_dict = ...

# Create cidr_format add it to the config dict
config = ...

# Create a bravado_core.spec.Spec
swagger_spec = Spec.from_dict(spec_dict, config=config)

# Get the operation to invoke
op = swagger_spec.get_op_for_request('GET', '/get_overlapping_cidrs')

# Get the Param that represents the cidr query parameter
cidr_param = op.params.get('cidr')

# Create a CIDR object - to_wire() will be called on this during marshalling
cidr_object = CIDR('192.168.1.1/24')
request_dict = {}

# Marshal the cidr_object into the request_dict.
marshal_param(cidr_param, cidr_object, request_dict)

# Lots of hand-wavey stuff here - use whatever http client you have to
# send the request and receive a response
response = http_client.send(request_dict)

# Extract the list of cidrs
cidrs = unmarshal_response(response)
```

(continues on next page)

(continued from previous page)

```
# Verify cidrs are CIDR objects and not strings
for cidr in cidrs:
    assert type(cidr) == CIDR
```

CHAPTER 4

Changelog

4.1 4.8.4 (2017-09-06)

- Make sure all models are properly tagged when flattening the spec - PR #191.

4.2 4.8.3 (2017-09-05)

- Improve spec flattening: recognize response objects and expose un-referenced models - PR #184.
- Fix a bug when marshalling properties with no spec that have the value `None` - PR #189.

4.3 4.8.2 (2017-09-04)

- Fix marshalling of `null` values for properties with `x-nullable` set to `true` - Issue #185, PR #186. Thanks Jan Baraniewski for the contribution!
- Add `_asdict()` method to each model, similar to what namedtuples have - PR #188.

4.4 4.8.1 (2017-08-24)

- Make unmarshalling objects roughly 30% faster - PR #182.

4.5 4.8.0 (2017-07-15)

- Add support for Swagger spec flattening - PR #177.

- Fix handling of API calls that return non-JSON content (specifically text content) - PR #175. Thanks mostrows2 for your contribution!
- Fix error message text when trying to unmarshal an invalid model - PR #179.

4.6 4.7.3 (2017-05-05)

- Fix support for object composition (allOf) for data passed in the request body - PR #167. Thanks Zi Li for your contribution!
- Return the default value for an optional field missing in the response - PR #171.

4.7 4.7.2 (2017-03-23)

- Fix unmarshalling of null values for properties with no spec - Issue #163, PR #165.

4.8 4.7.1 (2017-03-22)

- Fix backward-incompatible Model API change which renames all model methods to have a single underscore infront of them. A deprecation warning has been added - Issue #160, PR #161. Thanks Adam Ever-Hadani for the contribution!

4.9 4.7.0 (2017-03-21)

- Added support for nullable fields in the format validator - PR #143. Thanks Adam Ever-Hadani
- Add include_missing_properties configuration - PR #152
- Consider default when unmarshalling - PR #154
- Add discriminator support - PR #128, #159. Thanks Michael Jared Lumpe for your contribution
- Make sure pre-commit hooks are installed and run when running tests - PR #155, #158

4.10 4.6.1 (2017-02-15)

- Fix unmarshalling empty array types - PR #148
- Removed support for Python 2.6 - PR #147

4.11 4.6.0 (2016-11-28)

- Security Requirement validation (for ApiKey) - PR #124
- Allow self as name for model property, adds new “create” alternate model constructor - Issue #125, PR #126.
- Allow overriding of security specs - PR #121
- Adds minimal support for responses with text/* content_type.

4.12 4.5.1 (2016-09-27)

- Add marshal and unmarshal methods to models - PR #113, #120.

4.13 4.5.0 (2016-09-12)

- Support for model composition through the allOf property - Issue #7, PR #63, #110. Thanks David Bartle for the initial contribution!
- Fix issue with header parameter values being non-string types - PR #115.

4.14 4.4.0 (2016-08-26)

- Adds support for security scheme definitions, mostly focusing on the "apiKey" type - PR #112.

4.15 4.3.2 (2016-08-17)

- Fixes around unmarshalling, x-nullable and required behavior - Issue #108, PR #109. Big thanks to Zachary Roadhouse for the report and pull request!
- Fix AttributeError when trying to unmarshal a required array param that's not present - PR #111.

4.16 4.3.1 (2016-08-09)

- Check if a parameter is bool-type before assuming it's a string - PR #107. Thanks to Nick DiRienzo for the pull request!

4.17 4.3.0 (2016-08-04)

- Add support for `x-nullable` - Issue #47, PR #64 and #103. Thanks to Andreas Hug for the pull request!
- Fix support for vendor extensions at the path level - PR #95, #106. Thanks to Mikołaj Siedlarek for the initial pull request!

4.18 4.2.5 (2016-07-27)

- Add basepython python2.7 for flake8, docs, and coverage tox commands

4.19 4.2.4 (2016-07-26)

- coverage v4.2 was incompatible and was breaking the build. Added –append for the fix.

4.20 4.2.3 (2016-07-26)

- Accept tuples as a type list as well.

4.21 4.2.2 (2016-04-01)

- Fix marshalling of an optional array query parameter when not passed in the service call - PR #87

4.22 4.2.1 (2016-03-23)

- Fix optional enums in request params - Issue #77
- Fix resolving refs during validation - Issue #82

4.23 4.2.0 (2016-03-10)

- More robust handling of operationId which contains non-standard chars - PR #76
- Provide a client ingestible version of spec_dict with x-scope metadata removed. Accessible as Spec.client_spec_dict - Issue #78

4.24 4.1.0 (2016-03-01)

- Better handling of query parameters that don't have a value - Issue #68
- Allow marshalling of objects which are subclasses of dict - PR #61
- Fix boolean query params to support case-insensetive true/false and 0/1 - Issue #70
- Support for Swagger specs in yaml format - Issue #42
- Fix validation of server side request parameters when collectionFormat=multi and item type is not string - Issue #66
- Fix unmarshaling of server side request parameters when collectionFormat=multi and cardinality is one - PR #75

4.25 4.0.1 (2016-01-11)

- Fix unmarshalling of an optional array query parameter when not passed in the query string.

4.26 4.0.0 (2015-11-17)

- Support for recursive $refs - Issue #35
- Requires swagger-spec-validator 2.0.1

- Unqualified $refs no longer supported. Bad: `{"$ref": "User"}` Good: `{"$ref": "#/definitions/User"}`
- Automatic tagging of models is only supported in the root swagger spec file. If you have models defined in $ref targets that are in other files, you must manually tag them with 'x-model' for them to be available as python types. See Model Discovery for more info.

4.27 3.1.1 (2015-10-19)

- Fix the creation of operations that contain shared parameters for a given endpoint.

4.28 3.1.0 (2015-10-19)

- Added http `headers` to `bravado_core.response.IncomingResponse`.

4.29 3.0.2 (2015-10-12)

- Added docs on how to use user-defined formats.
- Added docs on how to configure bravado-core.
- *formats* added as a config option

4.30 3.0.1 (2015-10-09)

- Automatically tag models in external $refs - Issue #45 - see Model Discovery for more info.

4.31 3.0.0 (2015-10-07)

- User-defined formats are now scoped to a Swagger spec - Issue #50 (this is a non-backwards compatible change)
- Deprecated bravado_core.request.RequestLike and renamed to bravado_core.request.IncomingRequest
- Added *make docs* target and updated docs (still needs a lot of work though)

4.32 2.4.1 (2015-09-30)

- Fixed validation of user-defined formats - Issue #48

4.33 2.4.0 (2015-08-13)

- Support relative '$ref' external references in swagger.json
- Fix dereferencing of jsonref when given in a list

4.34 2.3.0 (2015-08-10)

- Raise MatchingResponseNotFound instead of SwaggerMappingError when a response can't be matched to the Swagger schema.

4.35 2.2.0 (2015-08-06)

- Add reason to IncomingResponse

4.36 2.1.0 (2015-07-17)

- Handle user defined formats for serialization and validation.

4.37 2.0.0 (2015-07-13)

- Move http invocation to bravado
- Fix unicode in model docstrings
- Require swagger-spec-validator 1.0.12 to pick up bug fixes

4.38 1.1.0 (2015-06-25)

- Better unicode support
- Python 3 support

4.39 1.0.0-rc2 (2015-06-01)

- Fixed file uploads when marshaling a request
- Renamed ResponseLike to IncomingResponse
- Fixed repr of a model when it has an attr with a unicode value

4.40 1.0.0-rc1 (2015-05-26)

- Use basePath when matching an operation to a request
- Refactored exception hierarchy
- Added use_models config option

4.41 0.1.0 (2015-05-13)

- Initial release

CHAPTER 5

Indices and tables

- genindex
- modindex
- search