web-transmute Documentation

Release 0.1

Yusuke Tsutsumi

Dec 06, 2017

Contents

transmute-core takes python functions, and converts them into APIs that include schema validation and documentation via swagger.

more specifically, transmute provides:

- declarative generation of http handler interfaces by parsing *python functions*.
- validation and serialization to and from a variety of content types (e.g. json or yaml).
- validation and serialization to and from native python objects which use schematics.
- *autodocumentation* of all handlers generated this way, via swagger.

transmute-core is the core library, containing shared functionality across framework-specific implementations.

Implementations exist for:

- aiohttp
- flask
- sanic

An example in flask looks like:

```
import flask_transmute
from flask import Flask

app = Flask(__name__)


# api GET method, path = /multiply
# take query arguments left and right which are integers, return an
# integer.
@flask_transmute.route(app, paths='/multiply')
@flask_transmute.annotate({"left": int, "right": int, "return": int})
def multiply(left, right):
    return left * right

# finally, you can add a swagger json and a swagger-ui page by:
flask_transmute.add_swagger(app, "/swagger.json", "/swagger")

app.run()
```

more specifically, transmute-core provides:

transmute-core is released under the MIT license.

However, transmute-core bundles swagger-ui with it, which is released under the Apache2 license.

To use this functionality, it's recommended to build or use a framework-specific wrapper library, to handle a more fluid integration.

If you are interested in creating a transmute library for your preferred web framework, please read *Creating a framework-specific library*. If you are interested in having it appear on this page, please send a PR against the core project.

User's Guide:

CHAPTER 1

Writing transmute-compatible functions

Note: this section is broadly applicable to all transmute frameworks.

Functions are converted to APIs by using an intermediary TransmuteFunction object.

A transmute function is identical to a standard Python function, with the addition of a few details:

1.1 Add function annotations for input type validation / documentation

if type validation and complete swagger documentation is desired, arguments should be annotated with types. For Python 3, function annotations are used.

For Python 2, transmute provides an annotate decorator:

```
import transmute_core

# the python 3 way
def add(left: int, right: int) -> int:
     return left + right

# the python 2 way
@transmute_core.annotate({"left": int, "right": int, "return": int})
def add(left, right):
    return left + right
```

By default, primitive types and schematics models are accepted. See *serialization* for more information.

1.2 Use transmute_core.describe to customize behaviour

Not every aspect of an api can be extracted from the function signature: often additional metadata is required. Transmute provides the "describe" decorator to specify those attributes.

```
import transmute_core  # these are usually also imparted into the
# top level module of the transmute

@transmute_core.describe(
    methods=["PUT", "POST"],  # the methods that the function is for
    # the source of the arguments are usually inferred from the method type, but can
    # be specified explicitly
    query_parameters=["blockRequest"],
    body_parameters=["name"]
    header_parameters=["authtoken"]
    path_parameters=["username"]
)
def create_record(name: str, blockRequest: bool, authtoken: str, username: str) ->␣
↪bool:
    if block_request:
        db.insert_record(name)
    else:
        db.async_insert_record(name)
    return True
```

1.3 Exceptions

By default, transmute functions only catch exceptions which extend `transmute_core.APIException`, which results in an http response with a non-200 status code. (400 by default):

```
from transmute_core import APIException

def my_api() -> int:
    if not retrieve_from_database():
        raise APIException(code=404)
```

However, many transmute frameworks allow the catching of additional exceptions, and converting them to an error response.

1.4 Query parameter arguments vs post parameter arguments

The convention in transmute is to have the method dictate the source of the argument:

- GET uses query parameters
- all other methods extract parameters from the body

This behaviour can be overridden with *transmute_core.decorators.describe*.

1.5 Additional Examples

1.5.1 Optional Values

transmute libraries support optional values by providing them as keyword arguments:

```
# count and page will be optional with default values,
# but query will be required.
def add(count: int=100, page: int=0, query: str) -> [str]:
    return db.query(query=query, page=page, count=count)
```

1.5.2 Custom Response Code

In the case where it desirable to override the default response code, the response_code parameter can be used:

```
@describe(success_code=201)
def create() -> bool:
    return True
```

1.5.3 Use a single schema for the body parameter

It's often desired to represent the body parameter as a single argument. That can be done using a string for body_parameters describe:

```
@describe(body_parameters="body", methods="POST"):
def submit_data(body: int) -> bool:
    return True
```

1.5.4 Multiple Response Types

To allow multiple response types, there is a combination of types that can be used:

```
from transmute_core import Response

@describe(paths="/api/v1/create_if_authorized/",
          response_types={
              401: {"type": str, "description": "unauthorized"},
              201: {"type": bool}
          })
@annotate({"username": str})
def create_if_authorized(username):
    if username != "im the boss":
        return Response("this is unauthorized!", 401)
    else:
        return Response(True, 201)
```

note that adding these will remove the documentation and type honoring for the default success result: it is assumed you will document all non-400 responses in the response_types dict yourself.

1.5.5 Headers in a Response

Headers within a response also require defining a custom response type:

```
from transmute_core import Response

@describe(paths="/api/v1/create_if_authorized/",
          response_types={
              200: {"type": str, "description": "success",
                    "headers": {
                        "location": {
                            "description": "url to the location",
                            "type": str
                        }
                    }
              },
          })
def return_url():
    return Response("success!", headers={
        "location": "http://foo"
    })
```

CHAPTER 2

Serialization

transmute-core provides a framework and default implementation to allow serializing objects to and from common data representation for an API. This is by chaining two parts:

1. object serialization to / from basic Python data types.
2. data types serialization to / from standard data notations (e.g. json or yaml)

out of the box, object serialization is supported for:

- bool
- float
- int
- str
- decimal.Decimal
- datetime.datetime
- lists, denoted by the form [<type] (deprecated)
- schematics models.
- **schematics types**
  - if a type class is passed, it will initialize the class to an instance.
- experimental support for attrs classes, specifically using cattrs's typed notation. This is temporary and will be replaced by attrs' type parameter.
- types denoted using the typing module (python 3.5.4+ and 3.6.2+ only)

2.1 Customization

`TransmuteContext`

Both of these components are customizable, either through passing a new TransmuteContext object, or modifying the default instance.

To learn more about customizing these serializers, please see the API reference for TransmuteContext, ObjectSerializer, and ContentTypeSerializer.

CHAPTER 3

Autodocumentation

For transmute routes, transmute-core provides functionality for documentation generation via Swagger. This is performed by reading the schematics model.

The swagger apis are dependent on the framework, so refer the documentation for that project. In general, a function to generate a swagger html page and a swagger.json is provided.

CHAPTER 4

Response

4.1 Response Shape

The response shape describes what sort of object is returned back by the HTTP response in cases of success.

4.1.1 Simple Shape

As of transmute-core 0.4.0, the default response shape is simply the object itself, serialized to the primitive content type. e.g.

```
from transmute_core import annotate
from schematics.models import Model
from schematics.types import StringType, IntType

class MyModel(Model):
    foo = StringType()
    bar = IntType()

@annotate("return": MyModel)
def return_mymodel():
    return MyModel({
        "foo": "foo",
        "bar": 3
    })
```

Would return the response

```
{
    "foo": "foo",
    "bar": 3
}
```

4.1.2 Complex Shape

Another common return shape is a nested object, contained inside a layer with details about the response:

```
{
    "code": 200,
    "success": true,
    "result": {
        "foo": "foo",
        "bar": 3
    }
}
```

This can be enabled by modifying the default context, or passing a custom one into your function:

```
from transmute_core import (
    default_context, ResponseShapeComplex,
    TransmuteContext
)

# modifying the global context, which should be done
# before any transmute functions are called.
default_context.response_shape = ResponseShapeComplex

# passing in a custom context
context = TransmuteContext(response_shape=ResponseShapeComplex)

transmute_route(app, fn, context=context)
```

4.1.3 Custom Shapes

Any class or object which implements *transmute_core.response_shape.ResponseShape* can be used as an argument to response_shape.

CHAPTER 5

TransmuteContext

To enable rapidly generating apis, transmute-core has embedded several defaults and decisions about technology choices (such as Schematics for schema validation).

The `TransmuteContext` allows customizing this behaviour. Transmute frameworks should allow one to provide and specify their own context by passing it as a keyword argument during a function call:

```
from flask_transmute import add_route

add_route(app, fn, context=my_custom_context)
```

It is also possible to modify transmute_core.default_context: this is the context that is referenced by all transmute functions by default.

CHAPTER 6

Creating a framework-specific library

6.1 Full framework in 100 statements or less

The reuse achieved in transmute-core has allowed the framework-specific libraries to be extremely thin: Initial integrations of Flask and aiohttp were achieved in less than 100 statements of python code.

If you find yourself writing a lot of code to integrate with transmute-core, consider sending an issue: there may be more functionality that can be contributed to the core to enable a thinner layer.

6.2 Overview

A transmute library should provide at a minimum the following functionality:

1. a way to convert a transmute_function to a handler for your framework of choice.
2. a way to register a transmute function to the application object
3. a way to generate a swagger.json from an application object

Reference implementations exist at:

- https://github.com/toumorokoshi/aiohttp-transmute
- https://github.com/toumorokoshi/flask-transmute

6.3 Simple Example

Here is a minimal implementation for Flask, clocking in at just under 200 lines including comments and formatting. (just under 100 without)

```
"""
An example integration with flask.
"""
import json
import sys
import transmute_core
from transmute_core import (
    describe, annotate,
    default_context,
    generate_swagger_html,
    get_swagger_static_root,
    ParamExtractor,
    SwaggerSpec,
    TransmuteFunction,
    NoArgument
)
from flask import Blueprint, Flask, Response, request
from functools import wraps

SWAGGER_ATTR_NAME = "_tranmute_swagger"
STATIC_PATH = "/_swagger/static"


def transmute_route(app, fn, context=default_context):
    """
    this is the main interface to transmute. It will handle
    adding converting the python function into the a flask-compatible route,
    and adding it to the application.
    """
    transmute_func = TransmuteFunction(fn)
    routes, handler = create_routes_and_handler(transmute_func, context)
    for r in routes:
        """
        the route being attached is a great place to start building up a
        swagger spec.  the SwaggerSpec object handles creating the
        swagger spec from transmute routes for you.

        almost all web frameworks provide some app-specific context
        that one can add values to. It's recommended to attach
        and retrieve the swagger spec from there.
        """
        if not hasattr(app, SWAGGER_ATTR_NAME):
            setattr(app, SWAGGER_ATTR_NAME, SwaggerSpec())
        swagger_obj = getattr(app, SWAGGER_ATTR_NAME)
        swagger_obj.add_func(transmute_func, context)
        app.route(r)(handler)


def create_routes_and_handler(transmute_func, context):
    """
    return back a handler that is the api generated
    from the transmute_func, and a list of routes
    it should be mounted to.
    """
    @wraps(transmute_func.raw_func)
    def handler():
        exc, result = None, None
        try:
```

```
            args, kwargs = ParamExtractorFlask().extract_params(
                context, transmute_func, request.content_type
            )
            result = transmute_func(*args, **kwargs)
        except Exception as e:
            exc = e
            """
            attaching the traceack is done for you in Python 3, but
            in Python 2 the __traceback__ must be
            attached to the object manually.
            """
            exc.__traceback__ = sys.exc_info()[2]
        """
        transmute_func.process_result handles converting
        the response from the function into the response body,
        the status code that should be returned, and the
        response content-type.
        """
        response = transmute_func.process_result(
            context, result, exc, request.content_type
        )
        return Response(
            response["body"],
            status=response["code"],
            mimetype=response["content-type"],
            headers=response["headers"]
        )
    return (
        _convert_paths_to_flask(transmute_func.paths),
        handler
    )


def _convert_paths_to_flask(transmute_paths):
    """
    convert transmute-core's path syntax (which uses {var} as the
    variable wildcard) into flask's <var>.
    """
    paths = []
    for p in transmute_paths:
        paths.append(p.replace("{", "<").replace("}", ">"))
    return paths


class ParamExtractorFlask(ParamExtractor):
    """
    The code that converts http parameters into function signature
    arguments is complex, so the abstract class ParamExtractor is
    provided as a convenience.

    override the methods to complete the class.
    """

    def __init__(self, *args, **kwargs):
        """
        in the case of flask, this is blank. But it's common
        to pass request-specific variables in the ParamExtractor,
        to be used in the methods.
```

```
        """
        super(ParamExtractorFlask, self).__init__(*args, **kwargs)

    def _get_framework_args(self):
        """
        this method should return back a dictionary of the values that
        are normally passed into the handler (e.g. the "request" object
        in aiohttp).

        in the case of flask, this is blank.
        """
        return {}

    @property
    @staticmethod
    def body():
        return request.get_data()

    @staticmethod
    def _query_argument(key, is_list):
        if key not in request.args:
            return NoArgument
        if is_list:
            return request.args.getlist(key)
        else:
            return request.args[key]

    @staticmethod
    def _header_argument(key):
        return request.headers.get(key, NoArgument)

    @staticmethod
    def _path_argument(key):
        return request.match_info.get(key, NoArgument)


def add_swagger(app, json_route, html_route, **kwargs):
    """
    add a swagger html page, and a swagger.json generated
    from the routes added to the app.
    """
    spec = getattr(app, SWAGGER_ATTR_NAME)
    if spec:
        spec = spec.swagger_definition(**kwargs)
    else:
        spec = {}
    encoded_spec = json.dumps(spec).encode("UTF-8")

    @app.route(json_route)
    def swagger():
        return Response(
            encoded_spec,
            # we allow CORS, so this can be requested at swagger.io
            headers={"Access-Control-Allow-Origin": "*"},
            content_type="application/json",
        )

    # add the statics
```

```
    static_root = get_swagger_static_root()
    swagger_body = generate_swagger_html(
        STATIC_PATH, json_route
    ).encode("utf-8")

    @app.route(html_route)
    def swagger_ui():
        return Response(swagger_body, content_type="text/html")

    # the blueprint work is the easiest way to integrate a static
    # directory into flask.
    blueprint = Blueprint('swagger', __name__, static_url_path=STATIC_PATH,
                          static_folder=static_root)
    app.register_blueprint(blueprint)

# example usage.


@describe(paths="/api/v1/multiply/{document_id}",
          header_parameters=["header"],
          body_parameters="foo")
@annotate({
    "left": int, "right": int, "header": int,
    "foo": str, "return": int, "document_id": str
})
def multiply(left, right, foo, document_id, header=0):
    return left * right


@describe(paths="/api/v1/multiply_body", body_parameters="body")
@annotate({"body": int})
def multiply_body(body):
    return left * right

@describe(paths="/api/v1/header",
          response_types={
              200: {"type": str, "description": "success",
                    "headers": {
                        "location": {
                            "description": "url to the location",
                            "type": str
                        }
                    }
              },
          })
def header():
    return transmute_core.Response(
        "foo", headers={"x-nothing": "value"}
    )

app = Flask(__name__)
transmute_route(app, multiply)
transmute_route(app, multiply_body)
transmute_route(app, header)
add_swagger(app, "/api/swagger.json", "/api/")

if __name__ == "__main__":
    app.run()
```

CHAPTER 7

Installing

transmute-core can be installed via Pip, from PyPI:

```
$ pip install transmute-core
```

Pip allows installation from source as well:

```
$ pip install git+https://github.com/toumorokoshi/transmute-core.git#egg=transmute-
↪core
```

API Reference:

CHAPTER 8

API Reference

8.1 TransmuteContext

class transmute_core.context.**TransmuteContext**(*serializers=None*, *content-type_serializers=None*, *response_shape=None*)

TransmuteContext contains all of the configuration points for a framework based off of transmute.

It is useful for customizing default behaviour in Transmute, such as serialization of additional content types, or using different serializers for objects to and from basic data times.

8.2 Decorators

transmute_core.decorators.**annotate**(*annotations*)

in python2, native annotions on parameters do not exist:

```
def foo(a : str, b: int) -> bool:
    ...
```

this provides a way to provide attribute annotations:

```
@annotate({"a": str, "b": int, "return": bool})
def foo(a, b):
    ...
```

transmute_core.decorators.**describe**(***kwargs*)

describe is a decorator to customize the rest API that transmute generates, such as choosing certain arguments to be query parameters or body parameters, or a different method.

Parameters

- **paths** (*list(str)*) – the path(s) for the handler to represent (using swagger's syntax for a path)

- **methods** (*list(str)*) – the methods this function should respond to. if non is set, transmute defaults to a GET.
- **query_parameters** (*list(str)*) – the names of arguments that should be query parameters. By default, all arguments are query_or path parameters for a GET request.
- **body_parameters** (*List[str] or str*) – the names of arguments that should be body parameters. By default, all arguments are either body or path parameters for a non-GET request.

  in the case of a single string, the whole body is validated against a single object.
- **header_parameters** (*list(str)*) – the arguments that should be passed into the header.
- **path_parameters** (*list(str)*) – the arguments that are specified by the path. By default, arguments that are found in the path are used first before the query_parameters and body_parameters.

8.3 Object Serialization

class transmute_core.object_serializers.**ObjectSerializer**

The object serializer is responsible for converting objects to and from basic data types. Basic data types are serializable to and from most common data representation languages (such as yaml or json)

Basic data types are:

- str (basestring in Python2, str in Python3)
- float
- int
- None
- dict
- list

The serializer decides what it can and can not serialize, and should raise an exception when a type it can not serialize is passed.

SchematicsSerializer is the default implementation used.

dump(*model*, *value*)

dump the value from a class to a basic datatype.

if the model or value is not valid, raise a SerializationException

load(*model*, *value*)

load the value from a basic datatype, into a class.

if the model or value is not valid, raise a SerializationException

to_json_schema(*model*)

return a dictionary representing a jsonschema for the model.

class transmute_core.object_serializers.**SchematicsSerializer**(*builtin_models=None*)

An ObjectSerializer which allows the serialization of basic types and schematics models.

The valid types that SchematicsSerializer supports are:

- int

- float
- bool
- decimal
- string
- none
- lists, in the form of [Type] (e.g. [str])
- any type that extends the schematics.models.Model.

8.4 ContentType Serialization

class transmute_core.contenttype_serializers.**ContentTypeSerializer**
A ContentTypeSerializer handles the conversion from a python data structure to a bytes object representing the content in a particular content type.

can_handle(*content_type_name*)
given a content type, returns true if this serializer can convert bodies of the given type.

content_type()
return back what a list of content types this serializer should support.

dump(*data*)
should return back a bytes (or string in python 2), representation of your object, to be used in e.g. response bodies.

a ValueError should be returned in the case where the object cannote be serialized.

load(*raw_bytes*)
given a bytes object, should return a base python data structure that represents the object.

a ValueError should be returned in the case where the object cannot be serialized.

main_type()
return back a single content type that represents this serializer.

class transmute_core.contenttype_serializers.**SerializerSet**(*serializer_list*)
composes multiple serializers, delegating commands to one that can handle the desired content type.

SerializerSet implements a dict-like interface. Retrieving serializers is done by get the content type item:

```
serializers["application/json"]
```

keys()
return a list of the content types this set supports.

this is not a complete list: serializers can accept more than one content type. However, it is a good representation of the class of content types supported.

class transmute_core.contenttype_serializers.**JsonSerializer**

static can_handle(*content_type_name*)
given a content type, returns true if this serializer can convert bodies of the given type.

static dump(*data*)
should return back a bytes (or string in python 2), representation of your object, to be used in e.g. response bodies.

static load(*raw_bytes*)
given a bytes object, should return a base python data structure that represents the object.

class transmute_core.contenttype_serializers.**YamlSerializer**

static can_handle(*content_type_name*)
given a content type, returns true if this serializer can convert bodies of the given type.

static dump(*data*)
should return back a bytes (or string in python 2), representation of your object, to be used in e.g. response bodies.

static load(*raw_bytes*)
given a bytes object, should return a base python data structure that represents the object.

8.5 Swagger

class transmute_core.swagger.**SwaggerSpec**
a class for aggregating and outputting swagger definitions, from transmute primitives

add_func(*transmute_func*, *transmute_context*)
add a transmute function's swagger definition to the spec

add_path(*path*, *path_item*)
for a given path, add the path items.

paths
return the path section of the final swagger spec, aggregated from the paths added.

swagger_definition(*title='example'*, *version='1.0'*, *base_path=None*)
return a valid swagger spec, with the values passed.

transmute_core.swagger.**generate_swagger_html**(*swagger_static_root*, *swagger_json_url*)
given a root directory for the swagger statics, and a swagger json path, return back a swagger html designed to use those values.

transmute_core.swagger.**get_swagger_static_root**()
transmute-core includes the statics to render a swagger page. Use this function to return the directory containing said statics.

8.6 Shape

class transmute_core.response_shape.**ResponseShape**
result shapes define the return format of the response.

static create_body(*result_dict*)
given the result dict from transmute_func, return back the response object.

static swagger(*result_schema*)
given the schema of the inner result object, return back the swagger schema representation.

class transmute_core.response_shape.**ResponseShapeComplex**
return back an object with the result nested, providing a little more context on the result:

- status code
- success

- result

class transmute_core.response_shape.**ResponseShapeSimple**
return back just the result object.

8.7 TransmuteFunction

Warning: transmute framework authors should not need to use attributes in TransmuteFunction directly. see *Creating a framework-specific library*

class transmute_core.function.transmute_function.**TransmuteFunction**(*func*, *args_not_from_request=None*)
TransmuteFunctions wrap a function and add metadata, allowing transmute frameworks to extract that information for their own use (such as web handler generation or automatic documentation)

get_response_by_code(*code*)
return the return type, by code

get_swagger_operation(*context=<transmute_core.context.TransmuteContext object>*)
get the swagger_schema operation representation.

process_result(*context*, *result_body*, *exc*, *content_type*)
given an result body and an exception object, return the appropriate result object, or raise an exception.

transmute_core.function.parameters.**get_parameters**(*signature*, *transmute_attrs*, *arguments_to_ignore=None*)
given a function, categorize which arguments should be passed by what types of parameters. The choices are:

- query parameters: passed in as query parameters in the url
- body parameters: retrieved from the request body (includes forms)
- header parameters: retrieved from the request header
- path parameters: retrieved from the uri path

The categorization is performed for an argument "arg" by:

1. examining the transmute parameters attached to the function (e.g. func.transmute_query_parameters), and checking if "arg" is mentioned. If so, it is added to the category.

2. If the argument is available in the path, it will be added as a path parameter.

3. If the method of the function is GET and only GET, then "arg" will be be added to the expected query parameters. Otherwise, "arg" will be added as a body parameter.

Changelog:

CHAPTER 9

Changelog

```
v1.6.0 (2017-11-03)
-------------------
- Modifying basePath back to being empty if not present. [Yusuke
  Tsutsumi]
- Fixing issue with path, header, and query parameters not copying
  additional type information. [Yusuke Tsutsumi]


v1.5.0 (2017-08-31)
-------------------
- Minor: updating docs. [Yusuke Tsutsumi]
- Specifying the minor version too. [Yusuke Tsutsumi]
- Disabling cattrs for < python3.5, for now. [Yusuke Tsutsumi]
- Removing jinja2 dependency. [Yusuke Tsutsumi]
- Minor: trying travis with python3.6. [Yusuke Tsutsumi]
- Minor: updating docs. [Yusuke Tsutsumi]
- Minor: updating documentation. [Yusuke Tsutsumi]
- Minor: adding validation for attrs jsonschema. [Yusuke Tsutsumi]
- Introducing support for cattrs. [Yusuke Tsutsumi]
- Migrating the converter to the old pattern. [Yusuke Tsutsumi]
- Implementing compatability with legacy serializer. [Yusuke Tsutsumi]
- Migrating all unit tests to execute against the serializer set.
  [Yusuke Tsutsumi]
- Adding new reqs. [Yusuke Tsutsumi]
- Integrating cattrs. [Yusuke Tsutsumi]
- Merge remote-tracking branch 'origin/attrs_support' [Yusuke Tsutsumi]
- Minor: cattrs does not support 3.6, switching to 3.6 for now. [Yusuke
  Tsutsumi]
- Minor: adding pypy checker, removing local develop install. [Yusuke
  Tsutsumi]
- Minor: adding a benchmark. [Yusuke Tsutsumi]
- Adding attrs-jsonschema for attrs serialization. [Yusuke Tsutsumi]
- Adding attrs support (#26) [Yun Xu]
```

```
v1.4.6 (2017-08-06)
-------------------
- Bugfix: instances of types were no longer considered serializable.
  [Yusuke Tsutsumi]
- Bugfix: supporting None, using basetype serializers instead of
  schematics serializers where applicable. [Yusuke Tsutsumi]
- Minor: adding unit tests. [Yusuke Tsutsumi]


v1.4.5 (2017-08-04)
-------------------
- Bugfix: correcting datetime type as per openapi spec. [Yusuke
  Tsutsumi]


v1.4.4 (2017-08-03)
-------------------
- Bugfix: datetime was not honored by schematics serializer. [Yusuke
  Tsutsumi]
- Bugfix: datetime was not returning proper json schema, nor a proper
  type. [Yusuke Tsutsumi]


v1.4.3 (2017-08-03)
-------------------
- Adding support for lists of types (fixes #29) [Yusuke Tsutsumi]


v1.4.2 (2017-08-02)
-------------------
- Provide a default content type, and handle none in the return type.
  fixes #27, #28. [Yusuke Tsutsumi]
- Minor: including notice about Apache2 license used by swagger-ui.
  [Yusuke Tsutsumi]
- Minor: updating documentation. [Yusuke Tsutsumi]
- Minor: adding basic tests for booltype. [Yusuke Tsutsumi]
- Implemented a serializer set for objects, to expand serialization
  options. [Yusuke Tsutsumi]
- Minor: adding sanic-transmute link. [Yusuke Tsutsumi]


v1.4.1 (2017-06-05)
-------------------
- Adding a check for serializable schematics types. [Yusuke Tsutsumi]
- Minor: surfacing undiscovered statics. (fixes #25) [Yusuke Tsutsumi]


v1.4.0 (2017-06-02)
-------------------
- Minor: fixing failing unit test with python3.6. [Yusuke Tsutsumi]
- Minor: adding documentation. [Yusuke Tsutsumi]
- Minor: adding pip selfcheck to gitignore. [Yusuke Tsutsumi]
- Adding the ability to document response headers. [Yusuke Tsutsumi]
- Minor: internal refactor to attrs. [Yusuke Tsutsumi]
- Adding a header field to the response object returned. [Yusuke
  Tsutsumi]
- Minor: modifying changelog regex. [Yusuke Tsutsumi]
- Minor: adding long string benchmark. [Yusuke Tsutsumi]
```

```
v1.3.2 (2017-05-29)
-------------------

Fix
~~~
- Body_parameters as a single argument was not properly being parsed.
  [Yusuke Tsutsumi]


v1.3.1 (2017-05-29)
-------------------

Fix
~~~
- UnboundLocalError when using path-only routes (#24) [Yusuke Tsutsumi]


v1.3.0 (2017-05-29)
-------------------
- Adding more testing around the scenario. [Yusuke Tsutsumi]
- Adding support for a blanket argument as the body parameter. [Yusuke
  Tsutsumi]


v1.2.0 (2017-05-23)
-------------------
- Enforcing compatibility with schematics 2.0.0 and new swagger-schema.
  [Yusuke Tsutsumi]


v1.1.1 (2017-05-16)
-------------------
- Bugfix: swagger body parameter did not adhere to swagger spec. [Yusuke
  Tsutsumi]

  fixes #11


v1.1.0 (2017-04-26)
-------------------
- Allowing different combinations of path parameters. [Yusuke Tsutsumi]


v1.0.1 (2017-04-20)
-------------------
- Fixing a bug with loading of schematics classes. [Yusuke Tsutsumi]


v1.0.0 (2017-04-19)
-------------------
- Support passing in a schematics type as well as an instance of a type
  object. [Yusuke Tsutsumi]


v0.6.0 (2017-03-22)
-------------------
```

```
- Merge pull request #9 from shwetas1205/master. [Yusuke Tsutsumi]

  OperationID support
- OperationID support. [Shweta Sharma]
- Handling conversionerror. [Yusuke Tsutsumi]


v0.4.12 (2017-03-14)
--------------------
- Bugfix: fixing traceback handling. [Yusuke Tsutsumi]


v0.4.11 (2017-03-14)
--------------------
- Including a facility to have clearer tracebacks from functions.
  [Yusuke Tsutsumi]


v0.5.0 (2017-02-10)
-------------------
- Adding the ability to create custom responses. [Yusuke Tsutsumi]


v0.4.10 (2017-01-20)
--------------------
- Packaging with correct swagger statics version. [Yusuke Tsutsumi]


v0.4.9 (2017-01-15)
-------------------
- Fix bug with only one swagger definition per path being honored (fixes
  toumorokoshi/flask-transmute#7) [Yusuke Tsutsumi]


v0.4.8 (2017-01-12)
-------------------
- Allowing content_type to be None when passed. [Yusuke Tsutsumi]


v0.4.7 (2017-01-09)
-------------------
- Adding datetime type to native types serializable. [Yusuke Tsutsumi]


v0.4.6 (2017-01-06)
-------------------
- Introducing add_path api to swagger object, for easier additions of
  new routes. [Yusuke Tsutsumi]


v0.4.5 (2017-01-04)
-------------------
- Updating swagger statics, adding summary strings. [Yusuke Tsutsumi]


v0.4.4 (2016-12-28)
-------------------
- Stricter checking around serialization types. [Yusuke Tsutsumi]
```

```
v0.4.3 (2016-12-28)
-------------------
- Fixing an issue with required parameters not documented properly in
  swagger. [Yusuke Tsutsumi]


v0.4.2 (2016-12-27)
-------------------

Fix
~~~
- Fixing complex shape return type. [Yusuke Tsutsumi]


v0.4.1 (2016-12-26)
-------------------
- Fixing bug with schematics schema not matching result object from
  response_shape. [Yusuke Tsutsumi]


v0.4.0 (2016-12-26)
-------------------
- Adding the ability to customize the response shape, and changing the
  default to simple. [Yusuke Tsutsumi]


v0.3.1 (2016-12-24)
-------------------
- Modifying method to add framework arguments, to defer logic to core.
  [Yusuke Tsutsumi]


v0.3.0 (2016-12-23)
-------------------
- Adding unit tests for new shared functionality. [Yusuke Tsutsumi]
- Adding a SwaggerSpec object, to help with swagger generation. [Yusuke
  Tsutsumi]


Warning: 'show' positional argument is deprecated.
```

Python Module Index

t

Index

R

S

T

Y