flask-transmute Documentation

Release 0.1

Yusuke Tsutsumi

Feb 16, 2018

Contents

A transmute framework for flask. This framework provides:

- declarative generation of http handler interfaces by parsing function annotations
- 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, using schematics.
- autodocumentation of all handlers generated this way, via swagger.

flask-transmute is provided as a flask extension, and can be included in the setup.py and/or requirements.txt for your service.

Here's a brief example:

```
import flask_transmute
from flask import Flask, Blueprint

app = Flask(__name__)
blueprint = Blueprint("blueprint", __name__, url_prefix="/foo")

# creates an api that:
# * accepts multiple markup types like json and yaml
# * validates with input types that are specified
@flask_transmute.route(app, paths='/multiple')
# annotate types to tell flask-transmute what to verify
# the type as (default string)
@flask_transmute.annotate({"left": int, "right": int, "return": int})
def multiply(left, right):
    return left * right

# if you use python 3.5+, you can annotate directly
# in the method signature.
@flask_transmute.route(app, paths='/multiply3')
def multiply_3(left: int, right: int) -> int:
    return left + right

# blueprints are supported as well
@flask_transmute.route(blueprint, paths='/subroute')
@flask_transmute.annotate({"return": bool})
def subroute():
    return True

app.register_blueprint(blueprint)

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

app.run()
```

CHAPTER 1

Legacy Implementation

flask-transmute 1.0 uses a completely different implementation of the transmute functionality based on transmute-core

Documentation for the pre-1.0 version can be found under the legacy section.

Contents:

1.1 Installing

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

```
$ pip install flask-transmute
```

Pip allows installation from source as well:

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

1.2 Routes

1.2.1 Example

Adding routes follows the flask pattern, with a decorator a decorator converting a function to a flask route

```
from flask_transmute import route, annotate

# define a GET endpoint, taking a query parameter integers left and right,
# which must be integers.
@route(app, paths="/multiply")
@annotate({"left": int, "right": int, "return": int})
```

```
def multiply(left, right):
    return left * right
```

see transmute-core:function for more information on customizing transmute routes.

1.2.2 API Documentation

1.3 Serialization

See serialization in transmute-core.

1.4 Autodocumentation

You can use add_swagger(app, json_path, html_path) to add swagger documentation for all transmute routes.

```
flask_transmute.add_swagger(app, "/swagger.json", "/swagger")
```

1.4.1 API Reference

1.5 pre 1.0 documentation

flask-transmute is a flask extension that generates APIs from standard python functions and classes. Autodocumention is also provided via swagger.

Here's a brief example:

```
import flask_transmute
from flask import Flask

route_set = flask_transmute.FlaskRouteSet()

class Pet(object):

    def __init__(self, name, classification):
        self.name = name
        self.classification = classification

    transmute_model = {
        "properties": {
            "name": {"type": str},
            "classification": {"type": str}
        },
        "required": ["name", "classification"]
    }

    @staticmethod
    def from_transmute_dict(model):
        return Pet(model["name"], model["classification"])

@route_set.route("/add_pet")
@flask_transmute.updates
```

```
# python 2 doesn't support parameter annotations.
# instead, you can do
# @flask_transmute.annotate({"pet": Pet, "return": Pet})
def add_pet(pet: Pet) -> Pet:
    animals.add(pet)
    return pet

app = Flask(__name__)
route_set.init_app(app)
```

The example above creates a path /add_pet that:

- accepts POST as a method (due to flask_transmute.updates)
- requires a body containing the fields "name" and "type"
- returns a response {"success": True, "response": True}

You can find a more in-depth example here: https://github.com/toumorokoshi/flask-transmute/blob/master/examples/deck.py

In raw flask, the above is equivalent to:

```
import json
from flask import Flask, jsonify, request

app = Flask(__name__)

class ApiException(Exception):
    pass

class Pet(object):

    def __init__(self, name, classification):
        self.name = name
        self.classification = classification

    @staticmethod
    def from_dict(model):
        return Pet(model["name"], model["classification"])

    @staticmethod
    def validate_pet_dict(model):
       errors = []

       if "name" not in model:
           errors.append("name not in model!")
       elif not isinstance(model["name"], str):
           errors.append("expected a string for name. found: {0}".format(type(model[
↪"name"])))

       if "classification" not in model:
           errors.append("name not in model!")
       elif not isinstance(model["classification"], str):
           errors.append("expected a string for classification. found: {0}".
↪format(type(model["classification"])))

       return errors

    def to_dict(self):
```

```
        return {"name": self.name, "classification": self.classification}

@app.route("/add_pet", methods=["POST"])
def add_pet():
    try:
        if "json" in request.content_type:
            request_args = json.loads(request.get_data().decode("UTF-8"))
        else:
            request_args = request.form

        if "pet" not in request_args:
            raise ApiException("pet field is required")

        errors = Pet.validate_pet_dict(request_args["pet"])

        if errors:
            raise ApiException(str(errors))

        pet = request_args["pet"]
        pet_object = Pet.from_dict(model)
        animals.add(pet_object)

        return jsonify({"success": True, "result": pet_object.to_dict()})
    except ApiException as e:
        return jsonify({"success": False, "detail": str(e)})
```

Contents:

1.5.1 autodocumentation

Autodocumention is generated via a swagger spec: swagger. It is generated by instatiating a swagger object, then using the "init_app" method:

```
from flask_transmute.swagger import Swagger
from flask import Flask

route_set = flask_transmute.FlaskRouteSet()

@route_set.route("/api/is_programming_fun")
def is_programming_fun(language: str) -> bool:
    """
    given a language, return True if programming
    in that language is fun.
    """
    return language.lower() == "python"


app = Flask(__name__)
route_set.init_app(app)
swagger = Swagger("myApi", "1.0", route_prefix="/api")
swagger.init_app(app)
```

init_app will attach two routes:

- <route_prefix>, which will be a swagger ui page with your route documentation.
- <route_prefix>/swagger.json, which will be the swagger spec itself.

1.5.2 decorators

flask-transmute attempts to extract as much information as possible from the function declaration itself. However, there are times when behaviour should be indicated that is not clear from the function signature.

to accomodate this, flask-transmute provides decorators to help describe additional behavior:

```
import flask_transmute

cards = []

# create indicates data is being created,
# correlated to a PUT request
@flask_transmute.creates
def create_card(card: str) -> bool:
     cards.append(card)
     return True

# updates indicates data is being updated,
# correlated to a POST request
@flask_transmute.updates
def update_card(old_card: str, new_card: str) -> bool:
    if old_card in cards:
        card_index = cards.index(old_card)
        cards.remove(old_card)
        cards.insert(card_index, new_card)
        return True
    return False

# deletes indicates data is being deleted,
# correlated to a DELETE request
@flask_transmute.deletes
def delete_card(card: str) -> bool:
    if card in cards:
        cards.remove(card)
        return True
    return False

# in Python 2, it is not possible to annotate functions. flask_transmute
# provides a decorator to help with that
@flask_transmute.creates
@flask_transume.annotates({"card": str, "return": bool})
def create_card(card):
    cards.append(card)
    return True
```

1.5.3 serialization

Due to the dynamic nature of Python objects, it's necessary to provide some attributes on class declarations that flask-transmute uses when serializing to and from a dictionary.

Two attributes are required:

- a dictionary attribute "transmute_schema" that specifies the structure of the object.
- a static method "from_transmute_dict" that receives a dictionary and should return an instance of the class.

The transmute_schema property is very similar to json-schemas, with the exception of using Python type objects instead of strings to define types:

```
transmute_schema = {
    "properties": {
        "cards": {
          "type": [Card]
        }
        "name": {"type": str}
    },
    "required": ["name"]
}
```

The following types may be used:

- [Type] (a list type)
- str
- bool
- NoneType
- int
- any class that is also serializable

The following keys are available:

- properties: this should be a dictionary of string keys and type declaration values, of the format {"type": cls}.
- required: this should be a list of attributes on the object that are required.

Here's a complete example:

```
class Card(object):

    def __init__(self, name, description):
        self.name = name
        self.description = description

    transmute_schema = {
        "properties": {
            "name": {"type": str},
            "description": {"type": str}
        },
        "required": ["name", "description"]
    }

    @staticmethod
    def from_transmute_dict(model):
        return Card(model["name"], model["description"])
```

1.5.4 Migrating to flask-transmute 1.0+

Migrating to the new transmute framework has multiple advantages:

- better swagger compliance
- customizable validation frameworks, default is Schematics
- up-to-date Swagger UI

- lack of dependencies on other unrelated libraries (such as flask-restful)

Migrating includes the following steps:

1. switch each model in transmute to schematics.
2. replace each FlaskRouteSet() with a blueprint
3. aggregating all decorators and routes to flask_transmute.route()
4. replace Swagger and swagger.init_app with add_swagger

The steps are outlined in detail below.

1. Converting to Schematics

flask-transmute has dropped it's proprietary json-schema based model definition, and adopted the Schematics library as the default serializer.

Note: the serializer is customizable. See TransmuteContext

---

flask_transmute.annotate() is still the correct function to use, to annotate rich types.

Primitive objects (str, int, float) are still supported. Complex objects should be represented as a Schematics schema instead:

```
# old style
class Pet(object):

    def __init__(self, name, classification):
        self.name = name
        self.classification = classification

    transmute_model = {
        "properties": {
            "name": {"type": str},
            "classification": {"type": str}
        },
        "required": ["name", "classification"]
    }

    @staticmethod
    def from_transmute_dict(model):
        return Pet(model["name"], model["classification"])

    @flask_transmute.annotate({"pet": Pet, "return": bool})
    def my_method(pet):
        return True
```

```
# new style
import flask_transmute
from schematics.models import Model
from schematics.types import StringType

class Pet(Model):
    name = StringType(required=True)
    classification = StringType(required=True)
```

```
@flask_transmute.annotate({"pet": Pet, "return": bool})
def my_method(pet):
    return True
```

2. Replace FlaskRouteSet with blueprints

flask-transmute now supports blueprints natively, and there is no longer a need for a custom object such as FlaskRouteSet. However, the flask_transmute.route should still be used over the standard blueprint.route:

```
# before
import flask_transmute
route_set = flask_transmute.FlaskRouteSet()

@route_set.route("/is_programming_fun")
@flask_transmute.updates
@flask_transmute.annotate({"answers": bool, "return": bool})
def is_programming_fun(answer):
    return True

route_set.init_app(app)
```

```
# after
from flask import Blueprint
import flask_transmute

blueprint = Blueprint("blueprint", __name__, url_prefix="/blueprint")

@flask_transmute.route(blueprint, paths="/is_programming_fun")
@flask_transmute.annotate({"answers": bool, "return": bool})
def is_programming_fun(answer):
    return True

app.register_blueprint(blueprint)
```

3. aggregate route descriptors to route

flask-transmute now aggregates all decorators into a single one: flask_transmute.describe. All arguments passed into the new flask_transmute.route are also passed along to a describe() call:

```
# before
@route_set.route("/is_programming_fun")
@flask_transmute.updates
@flask_transmute.annotate({"answers": bool, "return": bool})
def is_programming_fun(answer):
    return True
```

```
# after
@flask_transmute.route(app)
@flask_transmute.describe(paths="/is_programming_fun", methods=["POST"])
@flask_transmute.annotate({"answers": bool, "return": bool})
def is_programming_fun(answer):
    return True
```

Even simpler, arguments to describe can be passed into route directly:

```
# after
@flask_transmute.route(app, paths="/is_programming_fun", methods=["POST"])
@flask_transmute.annotate({"answers": bool, "return": bool})
def is_programming_fun(answer):
    return True
```

Warning: the new transmute syntax does not use the flask routing syntax, and uses the generic transmute-core path. Specifically, the path wildcard "/path/<var_name>" should be replaced with the wildcard "/path/{var_name}" instead.

4. replace init_swagger with add_swagger

Instead of instantiating and calling a swagger object, the add_swagger method should be used instead:

```
# before
from flask_transmute.swagger import Swagger

swagger = Swagger("myApi", "1.0", route_prefix="/api")
swagger.init_app(app)
```

```
# after
import flask_transmute

flask_transmute.add_swagger(app, "/api/swagger.json", "/api/",
                            title="myApi", version="1.0")
```

And you're done! You can learn more about how to customize in this document, and the transmute-core docs.

1.6 Indices and tables

- genindex
- modindex
- search