web-transmute Documentation

Release 0.1

Yusuke Tsutsumi

Mar 01, 2019

Contents

transmute-core removes the boilerplate of writing well-documented, easy to use APIs for Python web services, and easily integrates with any web framework. It takes a function that looks like this:

```
from transmute_core import annotate
from transmute_core.frameworks.flask import route

@route(app, paths='/multiply', tags=['math'])
def multiply(left: int, right: int) -> int:
    return left * right
```

Into an API /multiply that:

- validates and serializes objects into the proper object
- has an autodocumentation page for all APIs generated this way, via swagger.

The example above is for flask, but transmute-core has integrations for:

- aiohttp
- flask
- tornado

To learn more, see the *Getting Started*.

CHAPTER 1

License

transmute-core is released under the MIT license.

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

CHAPTER 2

User's Guide

2.1 Getting Started

This guide explains how to get the most out of transmute. These instructions have been written with flask in mind, but they apply to most frameworks with some minor tweaks.

For details that pertain to specifics framework, see framework support.

From a high level, the steps are:

1. authoring a function that you would like to be an API
2. annotating the function with data needed to describe how it should be exposed (method, path)
3. adding a route that exposes the swagger documentation page, displaying all transmute routes created

2.1.1 1. Authoring a Function To Be Transmuted

transmute-core is very flexible with regards to what sort of functions can be converted into APIs. The more best practices followed, the better.

Here's an ideal example:

```python
def multiply(left: int, right: int) -> int:
    """
    multiply two values together.
    """
    return left * right
```

transmute will extract relevant metadata about the function and use that to define certain attributes when creating the API. In the example above:

- multiply will allow two arguments: left and right, which are both integers
- the api will return back and integer

- the description of the api in documentation is “multiply two values together”

More complex object can be used. See [serialization](serialization.md).

Annotating Types in Python 2

The example above uses type annotations, which are only present in Python 3.4 and above. If you are using an older version, transmute-core provides the “annotate” decorator to provide the same data.

```
from transmute_core import annotate

@annotate({"left": int, "right": int, "return": int})
def multiply(left, right):
    """
    multiply two values together.
    """
    return left * right
```

2.1.2 2. Annotating a Function

This provides some of the data, but there is some missing information to fully define an API:

- the route that the API should be mounted to
- the method(s) that the API should respond to
- additional configuration, like where parameters should be found.

We can add that information with the describe function from transmute_core:

```
from transmute_core import annotate, describe

@describe(paths='/multiply', methods=["POST"])
def multiply(left: int, right: int) -> int:
    """
    multiply two values together.
    """
    return left * right
```

This specifies:

- the api should be mounted to the path /multiply
- multiply will response to the POST method
- since the method is POST, all arguments should be passed into the body

To attach the result to a flask application, transmute_core.frameworks.flask provides a route() function.

```
from transmute_core.frameworks.flask import route
from flask import Flask

app = Flask(__name__)

@route(app)
@describe(paths='/multiply', methods=["POST"])
def multiply(left: int, right: int) -> int:
    """
```

(continues on next page)

(continued from previous page)

```
    multiply two values together.
    """
    return left * right
```

As a shorthand, you can also pass configuration parameters into route as you would describe:

```
from transmute_core.frameworks.flask import route
from flask import Flask

app = Flask(__name__)

@route(app, paths='/multiply', methods=["POST"])
def multiply(left: int, right: int) -> int:
    """
    multiply two values together.
    """
    return left * right

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

At this point, you can start the server, and you can send it requests! Try it out:

```
$ curl http://localhost:8000/multiply --data='{"left": 10, "right": 20}'
```

But what about an easy way to view what APIs are available?

2.1.3 3. Adding Swagger Documentation to the App

As part of the route creation and mounting process, transmute will also add metadata that's easily discoverable. That metadata can be exposed as a swagger json payload. In addition, transmute-core bundles the swagger UI so you can view it easily and as a part of your application.

This is wrapped up as a single convenience method, provided per framework. For flask, it's transmute_core.frameworks.add_swagger:

```
from transmute_core.frameworks.flask import add_swagger

# note: this must be executed only after all APIs are mounted.
add_swagger(app, "/swagger.json", "/api/")
```

This mounts a the swagger json payload to /swagger.json, and provides a UI to view that at /api/.

At the end of the day, you can get a well documented API, and provide documentation, with roughly 4 lines from transmute_core.

```
from transmute_core.frameworks.flask import route, add_swagger
from flask import Flask

app = Flask(__name__)

@route(app, paths='/multiply', methods=["POST"])
def multiply(left: int, right: int) -> int:
    """
    multiply two values together.
```

(continues on next page)

(continued from previous page)

```
    """
    return left * right

add_swagger(app, "/swagger.json", "/api/")

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

Congrats! You have an application up.

2.1.4 4. What's Next?

You now have everything you need to get started with transmute! If you're interested in more complex objects in your apis, take a look at *Serialization and Validation*.

If you're looking for more complex use cases for the APIs such as specifying how parameters should be passed in, check out *Functions and Annotations*.

2.2 Serialization and Validation

As part of the API construction, tranmsute-core handles validating the incoming payload to match a schema, and serialization of the json or yaml payloads into native Python types.

transmute does not provide it's own schema definition or validation system. Instead, it hooks into several existing options, extracting json schemas and relying on their validation mechanism.

These types are matches to arguments using python type annotations, or the @transmure_core.annotate decorator.

At a high level, the following are supported:

- python primitive types
- attrs objects that use type annotations or have a type argument to attr.ib.
- types denoted using the typing module (installed as a separate package for python versions older than 3.5)
- schematics models and types.

The following discusses the types in detail.

2.2.1 Python Primitives

The following primitives can be used directly:

- bool
- float
- int
- str
- decimal
- datetime

2.2.2 Typing Module

Anything from the typing module is supported.

2.2.3 Attrs

Attrs provides a way to define types per attr.ib, which can be parsed by transmute. subtypes can be a combination of attrs objects, or the typing module. E.g. to define a list of attrs objects, you can use typing.List[MyAttrsClass].

2.2.4 Schematics

Both schematics models, and schematics types are supported.

(note: benchmarking has shows schematics to be very imperformant. See performance).

2.2.5 Details

Query Parameter Array Arguments as Comma-separated List

The intention is to ensure that serialization and deserialization of types matches that of openapi, to ensure that the UI provided is usable.

This forces behaviors such as multiple arguments being consumed as a comma-separated argument per query parameter, rather than just as multiple query parameter with the same name.

2.2.6 Performance

Among all of the components within transmute-core, the serialization and validation component has the most overhead (the rest are negligible relative to most application's business logic). As a result, the choice of object to use will have a huge impact on performance.

attrs is the most performant, with a huge con around error messages (a missing argument return back a wrong number of arguments passed into __init__).

schematics has great error messages, but is roughly 30x slower than attrs.

2.2.7 Customization

transmute-core can support additional object types, by modifying the global TransmuteContext object.

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.

2.3 Functions and Annotations

transmute-core infers a lot of data from the function metadata, but it's often necessary to express more complex scenarios.

This page discusses some details.

2.3.1 Argument Inference

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.describe.

2.3.2 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"],
    parameter_descriptions={
      "blockRequest": "if true, the request will be blocked.",
      "name": "the name of the db record to insert."
    }
)
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
```

2.3.3 Exceptions

By default, transmute functions only catch exceptions which extend transmute_core.APIException. When caught, the response is 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)
```

Many transmute frameworks allow the catching of additional exceptions, and converting them to an error response. See the framework specific guides for more details.

2.3.4 Additional Examples

Optional Values

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

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

Custom Response Code

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

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

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:

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

Multiple Response Types

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

```python
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.

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"
    })
```

2.4 Frameworks Supported

2.4.1 aiohttp

Concise Example

```
from aiohttp import web
from transmute_core.frameworks.aiohttp import (
    describe, add_swagger, route
)

@describe(paths="/multiply")
async def multiply(request, left: int, right: int) -> int:
    return left * right

app = web.Application()
route(app, multiply)
# this should be at the end, to ensure all routes are considered when
# constructing the handler.
add_swagger(app, "/swagger.json", "/swagger")
```

2.4.2 flask

Gotchas to Note

Flask normally using arrow quotes (<>) to specify variables in the path. Transmute-core uses the curly brackets ({}) to define path routes instead:

```
@route(app, paths='/{foo}')
def foo_path(foo: str) -> int:
    return 1
```

Concise Example

```
app = Flask(__name__)

@route(app, paths='/multiply', methods=["POST"])
def multiply(left: int, right: int) -> int:
    """
    multiply two values together.
    """
    return left * right

add_swagger(app, "/swagger.json", "/api/")

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

2.5 Advanced Usage

2.5.1 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.

2.5.2 Response

Response Shape

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

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()
```

(continues on next page)

(continued from previous page)

```python

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

Would return the response

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

Complex Shape

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

```json
{
    "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:

```python
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)
```

Custom Shapes

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

2.5.3 Creating a framework-specific library

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.

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

See transmute_core.frameworks for examples

Simple Example

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

```python
"""
An example integration with flask.
"""
import json
import sys
import transmute_core
import attr
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 schematics.models import Model
from schematics.types import StringType
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.
    """
```

(continues on next page)

(continued from previous page)

```
    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, methods=transmute_func.methods)(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 (
```

(continues on next page)

(continued from previous page)

```
        _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
    def body(self):
        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
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

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

@describe(paths="/api/v1/test")
@annotate({"vals": [int], "return": [int]})
def foo(vals):
    return vals


class SchematicsBody(Model):
    name = StringType(max_length=5)

@describe(paths="/api/v1/schematics",
          methods=["POST"],
          body_parameters="body")
@annotate({"body": SchematicsBody})
def schematics_example(body):
    return None

@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"}
    )

@attr.s
class AttrsExample(object):
    foo = attr.ib(type=str)

@describe(paths="/api/v1/attrs")
@annotate({"return": AttrsExample})
def attrs():
    return AttrsExample(foo="bar")

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

(continues on next page)

(continued from previous page)

```python

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

CHAPTER 3

API Reference

3.1 API Reference

3.1.1 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.

3.1.2 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.
- **parameter_descriptions** (*list (str)*) – descriptions for each parameter, keyed by attribute name. this will appear in the swagger documentation.

3.1.3 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.

`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.

3.1.4 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`()

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`()

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`()

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.

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

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.

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

3.1.5 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(*base_path=None*, ***kwargs*)
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.

3.1.6 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

`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.`**`ResponseShapeSimple`**

return back just the result object.

`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.

3.1.7 TransmuteFunction

Warning: transmute framework authors should not need to use attributes in TransmuteFunction directly. see creating_a_framework

`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:

3.2 Changelog

Command 'gitchangelog show HEAD...v0.2.9' failed: [Errno 2] No such file or directory: 'gitchangelog': 'gitchangelog'

Python Module Index

Index