swaggerconformance Documentation

Release 0.2.5

olipratt

Jun 27, 2018

Contents:

This is the documentation for the `swaggerconformance` package available on PyPI and Github.

- The *Introduction* outlines the purpose and setup of the package.
- The *Example Usage* walks through some standard use cases with example code.
- The *swaggerconformance API* documents the specifics of exposed API operations and their inputs and outputs.
- The Index and the search box in the sidebar are good places to start if you're looking for information on a specific object.

CHAPTER 1

swagger-conformance

You have a Swagger (aka OpenAPI) schema defining an API you provide - but does your API really conform to that schema, and does it correctly handle all valid inputs?

`swaggerconformance` combines the power of `hypothesis` for property based / fuzz testing with `pyswagger` to explore all corners of your API - testing its conformance to its specification.

1.1 Purpose

A Swagger/OpenAPI Spec allows you to carefully define what things are and aren't valid for your API to consume and produce. This tool takes that definition, and tries to make requests exploring all parts of the API while strictly adhering to the schema. Its aim is to find any places where your application fails to adhere to its own spec, or even just falls over entirely, so you can fix them up.

This is not a complete fuzz tester of your HTTP interface e.g. sending complete garbage, or to non-existent endpoints, etc. It's aiming to make sure that any valid client, using your API exactly as you specify, can't break it.

1.2 Setup

Either install with `pip install swagger-conformance`, or manually clone this repository and from inside it install dependencies with `pip install -r requirements.txt`.

1.3 Usage

After setup, the simplest test you can run against your API is just the following from the command line:

```
python -m swaggerconformance 'http://example.com/api/schema.json'
```

where the URL should resolve to your swagger schema, or it can be a path to the file on disk.

This basic test tries all your API operations looking for errors. For explanation of the results and running more thorough tests, including sequences of API calls and defining your custom data types, see the examples.

1.4 Documentation

Full documentation, including the example walkthroughs mentioned above and API documentation, is available here.

1.5 Wait, I don't get it, what does this thing do?

In short, it lets you generate example values for parameters to your Swagger API operations, make API requests using these values, and verify the responses.

For example, take the standard petstore API example. At the time of writing, that has an endpoint `/pet` with a `PUT` method operation that takes a relatively complicated `body` parameter.

With just a little code, we can load in the swagger schema for that API, access the operation we care about, and generate example parameters for that operation:

```
>>> import swaggerconformance
>>>
>>> client = swaggerconformance.client.Client('http://petstore.swagger.io/v2/swagger.
↪json')
>>>
>>> strategy_factory = swaggerconformance.strategies.StrategyFactory()
>>> operation = client.api.endpoints["/pet"]["put"]
>>> strategy = operation.parameters_strategy(strategy_factory)
>>> strategy.example()
{
  'body':{
    'id':110339,
    'name':'\U00052ea5\x9d\ua79d\x92\x13\U000f7c436!\U000aa3c5R\U0005b40e\n',
    'photoUrls':[
      '\ua9d9\U0003fb3a\x13\U00025c1c\U000974a8\u3497\U000515fa\n',
      "\U000b38a4>*\u6683'\U0002cd8f\x0f\n"
    ],
    'status':'sold',
    'category':{
      'id':-22555826027447
    },
    'tags':[
      {
        'id':-172930,
        'name':'\U000286df\u04dc\U00033563\u696d\U00055ba8\x89H'
      }
    ]
  }
}
>>>
```

See the examples for more details, and how to make requests against an API using these parameter values.

CHAPTER 2

Examples

Examples showing how to use this package.

2.1 Getting Started

The simplest way to test your API is just to ask `swaggerconformance` to validate it as follows (assuming your swagger spec is hosted at `http://example.com/api/schema.json` - a local file path could be used instead):

```
from swaggerconformance import api_conformance_test

api_conformance_test('http://example.com/api/schema.json')
```

This will simply make requests against your API using data generated to match your API schema, and ensure that the response codes and data returned also match your schema.

2.1.1 Handling Failures

If this testing fails, you'll be shown an example input to an API operation which resulted in a response that didn't match your schema. Something like:

```
Falsifying example: single_operation_test(
    client=SwaggerClient(schema_path='http://example.com/api/schema.json'),
    operation=OperationTemplate(
        method='get',
        path='/example/{exint}',
        params={'exint': ParameterTemplate(name='exint', type='integer',␣
↪required=True)}),
    params={'exint': -1})
...
<Some stack trace here>
...
AssertionError: Response code 500 not in {200, 404}
```

This shows that when testing the `GET` operation on the endpoint `/example/{exint}`, using the value `-1` for `exint` resulted in a `500` response code from the server, which is not one of the documented response codes (and any `5XX` response code is clearly an error!). From this we can tell that this server isn't handling negative numbers for this parameter properly and needs to be made more robust.

As an aside, one great feature of hypothesis is once it finds a failing example, it will simplify it down as far as it can. So here it might have first found that `-2147483648` failed, but instead of just reporting that and let you figure out if that number is special, it tries to find the simplest failing input value, e.g. here reducing down to simply `-1`.

2.2 Targeted and Stateful Testing

The basic testing above just throws data at your API, but it's useful to be able to target specific endpoints, or build sequenced tests where the output of one operation is used as the input to another.

2.2.1 Example

Here's a small example of a test that creates a resource with a `PUT` containing some generated data, and then verifies that a `GET` returns the exact same data.

You can run this test yourself by starting the `datastore_api.py` server and running the `ex_targeted_test.py` script in the examples directory. A walkthrough of the code there follows.

2.2.2 Walkthrough

The first part of the setup involves creating a client that will make requests against the API, and creating templates for all operations and parameters the API exposes. The client can be given a URL or local file path to read the schema from.

```
import hypothesis
import swaggerconformance

# Create the client to access the API and to template the API operations.
schema_url = 'http://127.0.0.1:5000/api/schema'
client = swaggerconformance.client.SwaggerClient(schema_url)
```

The next step pulls out the operations that will be tested, and creates `hypothesis` strategies for generating inputs to them. Here specific operations are accessed, but it's possible to just iterate through all operations in `client.api.template_operations()`. The `value_factory` generates strategies for individual data types being used - we'll just use the defaults, but the next section covers adding your own extra data types.

```
# Get references to the operations we'll use for testing, and strategies for
# generating inputs to those operations.
value_factory = swaggerconformance.valuetemplates.ValueFactory()
put_operation = client.api.endpoints["/apps/{appid}"]["put"]
put_strategy = put_operation.hypothesize_parameters(value_factory)
get_operation = client.api.endpoints["/apps/{appid}"]["get"]
get_strategy = get_operation.hypothesize_parameters(value_factory)
```

That's all the setup done - now to write the `hypothesis` test. The hypothesis docs go through details of doing this and the available test settings. In short though, you write a function which validates the property of your API you want to verify, and through a decorator `hypothesis` runs this function multiple times. Each attempt uses different parameter values chosen to test all corners of the allowed values - the more attempts you allow `hypothesis`, the more thorough it can be and the greater the chance it has to flush out bugs.

```
# Hypothesis will generate values fitting the strategies that define the
# parameters to the chosen operations. The decorated function is then called
# the chosen number of times with the generated parameters provided as the
# final arguments to the function.
@hypothesis.settings(max_examples=200)
@hypothesis.given(put_strategy, get_strategy)
def single_operation_test(client, put_operation, get_operation,
                          put_params, get_params):
    """PUT a new app with some data, then GET the data again and verify it
    matches what was just sent in."""
    # Use the client to make a request with the generated parameters, and
    # assert this returns a valid response.
    response = client.request(put_operation, put_params)
    assert response.status in put_operation.response_codes, \
        "{} not in {}".format(response.status, put_operation.response_codes)

    # The parameters are just dictionaries with string name keys and correct
    # format values.
    # We generated two independent sets of parameters, so replace the id in the
    # second set with the id from the first so they reference the same object.
    get_params["appid"] = put_params["appid"]
    response = client.request(get_operation, get_params)

    # `response.data` contains the JSON body of the response converted to
    # Python objects.
    # `payload` is always the name of body parameters on requests.
    out_data = response.data.data
    in_data = put_params["payload"]["data"]
    assert out_data == in_data, "{!r} != {!r}".format(out_data, in_data)

# Finally, remember to call the test function to run the tests, and that
# hypothesis provides the generated parameter arguments.
single_operation_test(client, put_operation, get_operation)
```

2.3 Custom Data Types

`swaggerconformance` supports generating values for all the standard datatypes from the OpenAPI schema. However you might have defined your own more specific ones and want to generate instances of them to get more thorough testing of valid inputs to your API.

2.3.1 Example

For example, suppose you have an API operation that takes a colour as a parameter, so you've defined a new datatype for your API as:

|Common Name|`type`|`format`|Comments| |———|——|———|——| |colour|`string`|`hexcolour`|# and six hex chars, e.g. `#3FE7D9`|

By default, clearly `swaggerconformance` won't know what this `format` is, and will just generate `string` type data as input for parameters of this format. So for most requests, your API will just rejecting them with some `4XX` response because the parameter isn't of the correct format. You might prefer that valid hex colours were being generated to test a greater number of successful API requests. New `type` and `format` support can be added into `swaggerconformance` fairly easily.

The first step here is to create a `ValueTemplate` which can build a hypothesis strategy to generate these `hexcolour` values:

```
import string
import hypothesis.strategies as hy_st
from swaggerconformance import valuetemplates

class HexColourTemplate(valuetemplates.ValueTemplate):
    """Template for a hex colour value."""

    def __init__(self, enum=None):
        # Various parameters like length don't make sense for this field, but
        # an enumeration of valid values may still be provided.
        super().__init__()
        self._enum = enum

    def hypothesize(self):
        # If there's a specific set of allowed values, the strategy should
        # return one of those.
        if self._enum is not None:
            return hy_st.sampled_from(self._enum)

        # We'll generate values as strings, but we could generate integers and
        # convert those to hex instead.
        strategy = hy_st.text(alphabet=set(string.hexdigits),
                              min_size=6,
                              max_size=6)
        # Don't forget to add the leading `#`.
        strategy = strategy.map(lambda x: "#" + x)

        return strategy
```

New `ValueTemplate`s just needs to provide a `hypothesize(self)` method which returns a `hypothesis` strategy - `__init__` is optional if no parameters are needed.

You can see the hypothesis docs for more information on building up strategies. One written, you can test values your template will produce - e.g.:

```
>>> template = HexColourTemplate()
>>> strategy = template.hypothesize()
>>> strategy.example()
'#CafB0b'
```

Now that the template for values of this type is defined, we just need to create an updated factory that will generate them. To do this, inherit from the built in one, and add logic into an overridden `create_value` method:

```
import swaggerconformance

class HexColourValueFactory(swaggerconformance.valuetemplates.ValueFactory):
    def create_value(self, swagger_definition):
        """Handle `hexcolour` string format, otherwise defer to parent class.
        :type swagger_definition:
            swaggerconformance.apitemplates.SwaggerParameter
        """
        if (swagger_definition.type == 'string' and
                swagger_definition.format == 'hexcolour'):
            return HexColourTemplate(swagger_definition.enum)
        else:
```

(continues on next page)

(continued from previous page)

```
            return super().create_value(swagger_definition)
```

Now whenever creating strategies for generating parameters for operations, use the new factory. Then anytime `string`, `hexcolour` is the datatype of a parameter, the new template will be used to generate a strategy for it. So in the example code in the previous section, the change would just be:

```
schema_url = 'http://example.com/api/schema.json'
client = swaggerconformance.client.SwaggerClient(schema_url)

value_factory = HexColourValueFactory()  # Use enhanced factory for values.
put_operation = client.api.endpoints["/apps/{appid}"]["put"]
put_strategy = put_operation.hypothesize_parameters(value_factory)
...
```

There's no reason that the factory couldn't check other properties of the parameter to decide what type of values to generate, e.g. the `name` - but remember that likely clients won't have whatever special logic you add here so it may be better to assign a special data type instead.

If needed, you can replace any of the built in value templates - e.g. to restrict the alphabet of all generated basic strings - but again remember that making the generation more restrictive might mean bugs are missed if clients don't have the same restrictions.

CHAPTER 3

swaggerconformance package

3.1 Subpackages

3.1.1 swaggerconformance.schema package

Module contents

This package provides templates which can be generated from a Swagger schema to represent parts of that schema which can later be filled in by generated values.

It also exposes the *Primitive* interface, which is the type of object that is passed to a `StrategyFactory` to generate values for.

class swaggerconformance.schema.**Api**(*client*)
Bases: object

Template for an entire Swagger API.

endpoints
Mapping of the endpoints of this API to their operations.

Return type dict(str, dict(str, *schema.Operation*))

operation(*operation_id*)
Access a single operation by its unique operation ID.

Return type *schema.Operation*

operations()
All operations of the API across all endpoints.

Return type Generator(*schema.Operation*)

class swaggerconformance.schema.**Operation**(*operation*)
Bases: object

Template for an operation on an endpoint.

Parameters **operation** (*pyswagger.spec.v2_0.objects.Operation*) – The definition of the operation in the API schema.

id
The Swagger operationId of this operation.

Return type str

method
The method of this operation.

Return type str

parameters
Mapping of the names of the parameters to their templates.

Return type dict(str, *Parameter*)

parameters_strategy(*value_factory*)
Generate hypothesis fixed dictionary mapping of parameters.

Parameters **value_factory** (strategies.StrategyFactory) – Factory to generate strategies for values.

path
The path of this operation.

Return type str

response_codes
List of HTTP response codes this operation might return.

Return type set(int)

class swaggerconformance.schema.**Parameter**(*swagger_definition*)
Bases: object

A Swagger API operation parameter.

Parameters **swagger_definition** (schema.Primitive) – The swagger spec portion defining the parameter.

format
The format of this parameter.

Return type str or None

name
The name of this parameter, if it has one.

Return type str or None

required
Whether this parameter is required.

Return type bool

strategy(*value_factory*)
Generate a hypothesis strategy representing this parameter.

Parameters **value_factory** (strategies.StrategyFactory) – Factory to generate strategies for values.

type
The type of this parameter.

Return type str

class swaggerconformance.schema.**Primitive**(*swagger_definition*)
Bases: object

Wrapper around a primitive in a swagger schema.

This may be a Parameter or a Schema Swagger object, either passed directly as a parameter to an operation as a child of one.

Since a Swagger Items object may be a child of a Parameter or schema, treat that the same as well since it's sufficiently similar we don't care about the distinction. Items don't have names though, so be careful of that.

Parameters **swagger_definition** (*pyswagger.spec.v2_0.objects.Parameter or pyswagger.spec.v2_0.objects.Items or pyswagger.spec.v2_0.objects.Schema*) – The swagger spec definition of this parameter.

additionalProperties
Whether this Primitive is a dict that accepts arbitrary entries.

Return type bool or None

enum
List of valid values for this Primitive.

Return type list or None

exclusiveMaximum
Whether the maximum value of this Primitive is allowed.

Return type bool

exclusiveMinimum
Whether the minimum value of this Primitive is allowed.

Return type bool

format
The format of this Primitive.

Return type str or None

items
The Parameter elements of this Primitive if it's an array.

Return type *Primitive* or None

location
The location of this Primitive - e.g. 'header' or 'body', or None if not a top-level primitive.

Return type str or None

maxItems
The maximum number of items in this Primitive if it's an array.

Return type int or None

maxLength
The maximum length of this Primitive.

Return type int or None

maxProperties
The maximum number of properties in this Primitive if it's a dict.

Return type int or None

maximum
The maximum value of this Primitive.

Return type float or None

minItems
The minimum number of items in this Primitive if it's an array.

Return type int or None

minLength
The minimum length of this Primitive.

Return type int or None

minProperties
The minimum number of properties in this Primitive if it's a dict.

Return type int or None

minimum
The minimum value of this Primitive.

Return type float or None

multipleOf
The value of this Primitive must be a multiple of this value.

Return type float or None

name
The name of this Primitive, if it has one.

Return type str or None

pattern
The regex pattern for this Primitive.

Return type string or None

properties
The dict of Primitive elements of this Primitive if it's an object.

Return type dict(str, *Primitive*) or None

required
Whether this Primitive is required.

Return type bool

required_properties
Set of required property names of this Primitive if it's an object.

Return type set(str) or None

type
The type of this Primitive.

Return type str

uniqueItems
Whether the items in this Primitive are unique if it's an array.

Return type bool

3.1.2 swaggerconformance.strategies package

Submodules

swaggerconformance.strategies.basestrategies module

Extra hypothesis strategies built from those in `hypothesis.strategies`, and helper functions for merging dictionary type strategies and dictionaries of strategies.

`swaggerconformance.strategies.basestrategies.`**`json`**(*value_limit=5*)
Hypothesis strategy for generating values that can be passed to `json.dumps` to produce valid JSON data.

Parameters **`value_limit`** (*int*) – A limit on the number of values in the JSON data - setting this too high can cause value generation to time out.

`swaggerconformance.strategies.basestrategies.`**`dates`**()
Hypothesis strategy for generating `datetime.date` values.

`swaggerconformance.strategies.basestrategies.`**`times`**()
Hypothesis strategy for generating `datetime.time` values.

`swaggerconformance.strategies.basestrategies.`**`datetimes`**()
Hypothesis strategy for generating `datetime.datetime` values.

`swaggerconformance.strategies.basestrategies.`**`file_objects`**()
Hypothesis strategy for generating pre-populated `file objects`.

`swaggerconformance.strategies.basestrategies.`**`files`**()
Hypothesis strategy for generating objects pyswagger can use as file handles to populate `file` format parameters.

Generated values take the format: `dict('data': <file object>)`

`swaggerconformance.strategies.basestrategies.`**`merge_dicts_strategy`**(*dict_strat_1*, *dict_strat_2*)
Strategy merging two strategies producting dicts into one.

`swaggerconformance.strategies.basestrategies.`**`merge_dicts_max_size_strategy`**(*dict1*, *dict2*, *max_size*)
Combine dict strategies into one to produce a dict up to a max size.

Assumes both dicts have distinct keys.

Parameters **`max_size`** (*int*) – Maximum number of keys in dicts generated by the strategy.

`swaggerconformance.strategies.basestrategies.`**`merge_optional_dict_strategy`**(*required_fields*, *optional_fields*)
Combine dicts of strings mapping to required and optional strategies.

Parameters

- **`required_fields`** (*dict(str)*) – Mapping containing required fields.
- **`optional_fields`** (*dict(str)*) – Mapping containing optional fields.

swaggerconformance.strategies.primitivestrategies module

Strategies for values of various data types.

class swaggerconformance.strategies.primitivestrategies.**PrimitiveStrategy**(*swagger_definition*, *factory*)

Bases: object

Strategy for a single value of any specified type.

Parameters

- **swagger_definition** (schema.Primitive) – The Swagger spec for this parameter.
- **factory** (strategies.StrategyFactory) – The factory used to generate child *PrimitiveStrategy* s.

strategy()

Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**BooleanStrategy**(*swagger_definition*, *factory*)

Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Strategy for a Boolean value.

strategy()

Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**NumericStrategy**(*swagger_definition*, *factory*)

Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Abstract template for a numeric value.

strategy()

Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**IntegerStrategy**(*swagger_definition*, *factory*)

Bases: *swaggerconformance.strategies.primitivestrategies.NumericStrategy*

Strategy for an integer value.

strategy()

Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**FloatStrategy**(*swagger_definition*, *factory*)

Bases: *swaggerconformance.strategies.primitivestrategies.NumericStrategy*

Strategy for a floating point value.

strategy()

Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**StringStrategy**(*swagger_definition*, *factory*, *blacklist_chars=None*)

Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Strategy for a string value.

strategy()

Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**URLPathStringStrategy**(*swagger_definition*, *factory*)

Bases: *swaggerconformance.strategies.primitivestrategies.StringStrategy*

Strategy for a string value which must be valid in a URL path.

class swaggerconformance.strategies.primitivestrategies.**HTTPHeaderStringStrategy**(*swagger_definition*, *factory*)

Bases: *swaggerconformance.strategies.primitivestrategies.StringStrategy*

Strategy for a string value which must be valid in a HTTP header.

strategy()

Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**XFieldsHeaderStringStrategy**(*swagger_definition*, *factory*)

Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Strategy for a string value which must be valid in the X-Fields header.

The `X-Fields` parameter lets you specify a mask of fields to be returned by the application. The format is a comma-separated list of fields to return, enclosed by curly-brackets, which can be nested. So for example: `{name, age, pets{name}}`. There is also a special value of `*` meaning 'all remaining fields', and it can be provided alone as `*` or inserted into a mask of the above format instead of a field name.

We could generate random values for this header that match the allowed syntax, but:

- as far as I can see, this field is a `Flast-RESTPlus` special,
- this is implemented by the Swagger API framework, so not exciting to exercise,
- there's a risk of just generating values which are rejected continually and so reducing the effectiveness of the testing of other fields.

Therefore, we just return either `''` or `*` for this parameter as they are safe values that shouldn't interfere with other testing.

strategy()

Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**DateStrategy**(*swagger_definition*, *factory*)

Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Strategy for a Date value.

strategy()
Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**DateTimeStrategy**(*swagger_definition*, *factory*)
Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Strategy for a Date-Time value.

strategy()
Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**UUIDStrategy**(*swagger_definition*, *factory*)
Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Strategy for a UUID value.

strategy()
Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**FileStrategy**(*swagger_definition*, *factory*)
Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Strategy for a File value.

strategy()
Return a hypothesis strategy defining this value.

class swaggerconformance.strategies.primitivestrategies.**ArrayStrategy**(*swagger_definition*, *factory*)
Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Strategy for an array collection.

strategy()
Return a hypothesis strategy defining this collection.

class swaggerconformance.strategies.primitivestrategies.**ObjectStrategy**(*swagger_definition*, *factory*)
Bases: *swaggerconformance.strategies.primitivestrategies.PrimitiveStrategy*

Strategy for a JSON object collection.

MAX_ADDITIONAL_PROPERTIES is a limit on the number of additional properties to add to objects. Setting this too high might cause data generation to time out.

MAX_ADDITIONAL_PROPERTIES = 5

strategy()
Return a hypothesis strategy defining this collection, including random additional properties if the object supports them.

Will add only up to *MAX_ADDITIONAL_PROPERTIES* extra values to prevent data generation taking too long and timing out.

Parameters

- **required_properties** (*dict*) – The required fields in the generated dict.

- **optional_properties** (*dict*) – The optional fields in the generated dict.

Module contents

This package provides access to two main classes of objects:

- *StrategyFactory* for generating `PrimitiveStrategy` instances matching the parameter values defines by a parameter entry in a swagger schema. This can be inherited from to generate new `PrimitiveStrategy` instances matching custom parameters in a user's schema.
- The `PrimitiveStrategy` itself and child classes used to generate hypothesis strategies for generating parameter values with certain constraints. Again, users may create new `PrimitiveStrategy` subclasses to define their own new value types.

class swaggerconformance.strategies.**StrategyFactory**
Bases: `object`

Factory for building `PrimitiveStrategy` from swagger definitions.

produce(*swagger_definition*)
Create a template for the value specified by the definition.

Parameters **swagger_definition** (`schema.Primitive`) – The schema of the parameter to create.

Return type *PrimitiveStrategy*

register(*type_str*, *format_str*, *creator*)
Register a function to generate `PrimitiveStrategy` instances for this type and format pair.

The function signature of the `creator` parameter must be:

```
def fn(schema.Primitive, StrategyFactory) -> PrimitiveStrategy
```

Parameters

- **type_str** (*str*) – The Swagger schema type to register for.
- **format_str** (*str*) – The Swagger schema format to register for.
- **creator** (*callable*) – The function to create a `PrimitiveStrategy`.

register_type_default(*type_str*, *creator*)
Register a function to generate `PrimitiveStrategy` instances for this type paired with any format with no other registered creator.

The function signature of the `creator` parameter must be:

```
def fn(schema.Primitive, StrategyFactory) -> PrimitiveStrategy
```

Parameters

- **type_str** (*str*) – The Swagger schema type to register for.
- **creator** (*callable*) – The function to create a `PrimitiveStrategy`.

swaggerconformance.strategies.**string_primitive_strategy**(*swagger_definition*, *factory*)
Function for creating an appropriately formatted string value depending on the location of the string parameter.

Parameters **swagger_definition** (`schema.Primitive`) – The schema of the parameter to create.

Return type *PrimitiveStrategy*

3.2 Submodules

3.2.1 swaggerconformance.client module

A client for accessing a remote swagger-defined API.

class swaggerconformance.client.**Client**(*schema_path*, *codec=None*)
Bases: object

Client to use to access the Swagger application according to its schema.

Parameters

- **schema_path** (*str*) – The URL of or file path to the API definition.
- **codec** (codec.CodecFactory *or None*) – Used to convert between JSON and objects.

api
The API accessible from this client.

Return type schema.Api

request(*operation*, *parameters*)
Make a request against a certain operation on the API.

Parameters

- **operation** (schema.Operation) – The operation to perform.
- **parameters** (*dict*) – The parameters to use on the operation.

Return type pyswagger.io.Response

3.2.2 swaggerconformance.codec module

Provides a codec for converting between JSON representations of objects and the objects themselves.

class swaggerconformance.codec.**CodecFactory**
Bases: object

Produces codecs that encode objects as JSON and decode JSON back into objects.

produce(*swagger_definition*, *value*)
Construct an object with the given value, represented by the given schema portion using the registered type/format mappings.

Parameters

- **swagger_definition** (schema.Primitive) – The Swagger schema type to register for.
- **value** – The value to use to build the object - may be the applicable portion of JSON after json.loads processing, or any supported input value for the relevant object.

register(*type_str*, *format_str*, *creator*)
Register a new creator for objects of the given type and format.

The creator parameter must be a callable which takes the following paramters in order:

- schema.Primitive - the Swagger schema for the object being handled.

- The value to use to build the object - may be the applicable portion of JSON after `json.loads` processing, or any supported input value for the relevant object.
- *`CodecFactory`* - this factory, to be used to generate child objects if required, by calling the `produce` method on it.

Parameters

- **`type_str`** (*`str`*) – The Swagger schema type to register for.
- **`format_str`** (*`str`*) – The Swagger schema format to register for.
- **`creator`** (*`callable`*) – The callable to create an object of the desired type.

3.2.3 swaggerconformance.response module

A response received to a Swagger API operation.

class `swaggerconformance.response.`**Response**(*raw_response*)

Bases: `object`

A response received to a Swagger API operation.

Parameters **`raw_response`** (*`pyswagger.io.Response`*) – The raw response.

`body`

Parsed response body converted to objects via the codec in use.

`headers`

HTTP headers received on the response.

Example format is `{'Content-Type': [xxx, xxx]}`

Header field names are case insensitive (See http://www.ietf.org/rfc/rfc2616.txt)

Return type dict(str, list(str))

`raw`

Raw response body.

Return type bytes

`status`

HTTP status code of the response.

Return type int

3.3 Module contents

This package enables easy testing of Swagger Spec defined APIs using property based tests and by generating paramter values meeting the specification.

This top-level package exposes functions for basic API tests just requiring access to the API schema.

Subpackages and modules then provide classes and functions for finer grain control over value generation and test procedures.

`swaggerconformance.`**`api_conformance_test`**(*schema_path*, *num_tests_per_op=20*, *cont_on_err=True*)

Basic test of the conformance of the API defined by the given schema.

Parameters

- **schema_path** (*str*) – The path to / URL of the schema to validate.
- **num_tests_per_op** (*int*) – How many tests to run of each API operation.
- **cont_on_err** (*bool*) – Validate all operations, or drop out on first error.

swaggerconformance.**operation_conformance_test**(*client*, *operation*, *num_tests=20*)

Test the conformance of the given operation using the provided client.

Parameters

- **client** (client.Client) – The client to use to access the API.
- **operation** (schema.Operation) – The operation to test.
- **num_tests** (*int*) – How many tests to run of each API operation.

Python Module Index

S

Index

N

O

P

R

S

T

U

X