Python JSONSchema Objects Documentation

Release 0.0.18

Chris Wacek

Oct 07, 2017

Contents

python-jsonschema-objects provides an *automatic* class-based binding to JSON schemas for use in python.

CHAPTER 1

What

python-jsonschema-objects provides an *automatic* class-based binding to JSON schemas for use in python.

For example, given the following schema:

```
{
    "title": "Example Schema",
    "type": "object",
    "properties": {
        "firstName": {
            "type": "string"
        },
        "lastName": {
            "type": "string"
        },
        "age": {
            "description": "Age in years",
            "type": "integer",
            "minimum": 0
        },
        "dogs": {
            "type": "array",
            "items": {"type": "string"},
            "maxItems": 4
        },
        "address": {
            "type": "object",
            "properties": {
                "street": {"type": "string"},
                "city": {"type": "string"},
                "state": {"type": "string"}
                },
            "required":["street", "city"]
            },
        "gender": {
            "type": "string",
```

```
            "enum": ["male", "female"]
        },
        "deceased": {
            "enum": ["yes", "no", 1, 0, "true", "false"]
            }
    },
    "required": ["firstName", "lastName"]
}
```

jsonschema-objects can generate a class based binding. Assume here that the schema above has been loaded in a variable called `schema`:

```
>>> import python_jsonschema_objects as pjs
>>> builder = pjs.ObjectBuilder(schema)
>>> ns = builder.build_classes()
>>> Person = ns.ExampleSchema
>>> james = Person(firstName="James", lastName="Bond")
>>> james.lastName
u'Bond'
>>> james
<example_schema lastName=Bond age=None firstName=James>
```

Validations will also be applied as the object is manipulated.

```
>>> james.age = -2
python_jsonschema_objects.validators.ValidationError: -2 was less
or equal to than 0
```

The object can be serialized out to JSON:

```
>>> james.serialize()
'{"lastName": "Bond", "age": null, "firstName": "James"}'
```

CHAPTER 2

Why

Ever struggled with how to define message formats? Been frustrated by the difficulty of keeping documentation and message definition in lockstep? Me too.

There are lots of tools designed to help define JSON object formats, foremost among them JSON Schema. JSON Schema allows you to define JSON object formats, complete with validations.

However, JSON Schema is language agnostic. It validates encoded JSON directly - using it still requires an object binding in whatever language we use. Often writing the binding is just as tedious as writing the schema itself.

This avoids that problem by auto-generating classes, complete with validation, directly from an input JSON schema. These classes can seamlessly encode back and forth to JSON valid according to the schema.

CHAPTER 3

Other Features

The ObjectBuilder can be passed a dictionary specifying 'memory' schemas when instantiated. This will allow it to resolve references where the referenced schemas are retrieved out of band and provided at instantiation.

For instance:

```
{
    "title": "Address",
    "type": "string"
}
```

```
{
    "title": "AddlPropsAllowed",
    "type": "object",
    "additionalProperties": true
}
```

```
{
    "title": "Other",
    "type": "object",
    "properties": {
        "MyAddress": {"$ref": "memory:Address"}
    },
    "additionalProperties": false
}
```

Generated wrappers can also properly deserialize data representing 'oneOf' relationships, so long as the candidate schemas are unique.

```
{
    "title": "Age",
    "type": "integer"
}
```

```
{
    "title": "OneOf",
    "type": "object",
    "properties": {
        "MyData": { "oneOf":[
            {"$ref": "memory:Address"},
            {"$ref": "memory:Age"}
            ]
        }
    },
    "additionalProperties": false
}
```

```
{
    "title": "OneOfBare",
    "type": "object",
    "oneOf":[
            {"$ref": "memory:Other"},
            {"$ref": "memory:Example Schema"}
            ],
    "additionalProperties": false
}
```

CHAPTER 4

Generating Multiple Top Level Objects

Sometimes what you really want to do is define a couple of different objects in a schema, and then be able to use them flexibly.

Any object built as a reference can be obtained from the top level namespace. Thus, to obtain multiple top level classes, define them separately in a definitions structure, then simply make the top level schema refer to each of them as a `oneOf`.

The schema and code example below show how this works.

```
{
    "title": "MultipleObjects",
    "id": "foo",
    "type": "object",
    "oneOf":[
            {"$ref": "#/definitions/ErrorResponse"},
            {"$ref": "#/definitions/VersionGetResponse"}
            ],
    "definitions": {
        "ErrorResponse": {
            "title": "Error Response",
            "id": "Error Response",
            "type": "object",
            "properties": {
                "message": {"type": "string"},
                "status": {"type": "integer"}
            },
            "required": ["message", "status"]
        },
        "VersionGetResponse": {
            "title": "Version Get Response",
            "type": "object",
            "properties": {
                "local": {"type": "boolean"},
                "version": {"type": "string"}
            },
```

```
            "required": ["version"]
        }
    }
}
```

```
>>> builder = pjs.ObjectBuilder('multiple_objects.json')
>>> classes = builder.build_classes()
>>> print(dir(classes))
[u'ErrorResponse', 'Local', 'Message', u'Multipleobjects',
'Status', 'Version', u'VersionGetResponse']
```

CHAPTER 5

Installation

```
pip install python_jsonschema_objects
```

CHAPTER 6

Tests

Tests are managed using the excellent Tox. Simply `pip install tox`, then `tox`.

CHAPTER 7

Changelog

0.0.18

- Fix assignment to schemas defined using 'oneOf'
- Add sphinx documentation and support for readthedocs

0.0.16 - Fix behavior of exclusiveMinimum and exclusiveMaximum validators so that they work properly.

0.0.14 - Roll in a number of fixes from Github contributors, including fixes for oneOf handling, array validation, and Python 3 support.

0.0.13 - Lazily build object classes. Allows low-overhead use of jsonschema validators.

0.0.12 - Support "true" as a value for 'additionalProperties'

0.0.11 - Generated wrappers can now properly deserialize data representing 'oneOf' relationships, so long as the candidate schemas are unique.

0.0.10 - Fixed incorrect checking of enumerations which previously enforced that all enumeration values be of the same type.

0.0.9 - Added support for 'memory:' schema URIs, which can be used to reference externally resolved schemas.

0.0.8 - Fixed bugs that occurred when the same class was read from different locations in the schema, and thus had a different URI

0.0.7 - Required properties containing the '@' symbol no longer cause `build_classes()` to fail.

0.0.6 - All literals now use a standardized LiteralValue type. Array validation actually coerces element types. `as_dict` can translate objects to dictionaries seamlessly.

0.0.5 - Improved validation for additionalItems (and tests to match). Provided dictionary-syntax access to object properties and iteration over properties.

0.0.4 - Fixed some bugs that only showed up under specific schema layouts, including one which forced remote lookups for schema-local references.

0.0.3b - Fixed ReStructuredText generation

0.0.3 - Added support for other array validations (minItems, maxItems, uniqueItems).

0.0.2 - Array item type validation now works. Specifying 'items', will now enforce types, both in the tuple and list syntaxes.

0.0.1 - Class generation works, including 'oneOf' and 'allOf' relationships. All basic validations work.

CHAPTER 8

API Documentation

Generated Classes

Classes generated using `python_jsonschema_objects` expose all defined properties as both attributes and through dictionary access.

In addition, classes contain a number of utility methods for serialization, deserialization, and validation.

class `python_jsonschema_objects.classbuilder.`**`ProtocolBase`**(***props*)

An instance of a class generated from the provided schema. All properties will be validated according to the definitions provided. However, whether or not all required properties have been provide will *not* be validated.

Parameters **`**props`** – Properties with which to populate the class object

Returns The class object populated with values

Raises `validators.ValidationError` – If any of the provided properties do not pass validation

`as_dict`()

Return a dictionary containing the current values of the object.

Returns The object represented as a dictionary

Return type (dict)

classmethod **`from_json`**(*jsonmsg*)

Create an object directly from a JSON string.

Applies general validation after creating the object to check whether all required fields are present.

Parameters **`jsonmsg`** (*`str`*) – An object encoded as a JSON string

Returns An object of the generated type

Raises `ValidationError` – if *jsonmsg* does not match the schema *cls* was generated from

`validate`()

Applies all defined validation to the current state of the object, and raises an error if they are not all met.

Raises `ValidationError` – if validations do not pass

Index