Sphinx JSON Schema Documentation

Release 1.17.2

Leo Noordergraaf

Apr 20, 2022

Contents

This Sphinx extension allows authors to embed a JSON Schema in their documentation.

It arose out of a personal itch and implements what I needed. Some features of JSON Schema are (not yet) implemented. Also I can imagine that other display layouts are desired.

I only tested it for use with the draft 4 specification of JSON Schema. I was pleasantly surprised to find that the software is useful to others as well. Therefore it made sense to document its intended use.

CHAPTER 1

Contents

1.1 Installation

Obtain sphinx-jsonschema by installing it with pip:

```
sudo pip install sphinx-jsonschema
```

Then add it to your project by editing the `conf.py` file and append 'sphinx-jsonschema' to the `extensions` array.

```
extensions = [
    'sphinx.ext.autodoc',
    'sphinx-jsonschema'
]
```

1.1.1 Source code

The source code for this extension can be found on GitHub.

1.1.2 Docker image

A Docker image containing Sphinx and a number of extensions, including sphinx-jsonschema, can be found at Extended Sphinx. This Docker image is generated from the Dockerfile on Github.

1.2 Directive

The extension adds a single directive to Sphinx: **jsonschema**. You provide it with either a file name, an HTTP(S) URL to a schema or you may embed the schema inline.

The schemas are read by a YAML parser. This means that you can write the schemas in either json or yaml notation and they will be processed identically.

1.2.1 Usage

To display a schema fetched from a website:

```
.. jsonschema:: http://example.com/project/schema.json
```

To display a schema in a file referenced by an absolute path use:

```
.. jsonschema:: /var/www/project/schema.json
```

or with a path relative to the current document:

```
.. jsonschema:: schemas/sample.json
```

this assumes that next to the .rst file containing the above statement there is a subdirectory `schemas` containing `sample.json`.

With any of the above references you can use JSON Pointer notation to display a subschema:

```
.. jsonschema:: http://example.com/project/schema.json#/definitions/sample

.. jsonschema:: /var/www/project/schema.json#/definitions/sample

.. jsonschema:: schemas/sample.json#/definitions/sample
```

Alternatively you can embed the schema directly into your documentation:

```
.. jsonschema::

    {
        "$schema": "http://json-schema.org/draft-04/schema#",
        "title": "An example",
        "id": "http://example.com/schemas/example.json",
        "description": "This is just a tiny example of a schema rendered by `sphinx-
↪jsonschema <http://github.com/lnoor/sphinx-jsonschema>`_.\n\nYes that's right you␣
↪can use *reStructuredText* in a description.",
        "type": "string",
        "minLength": 10,
        "maxLength": 100,
        "pattern": "^[A-Z]+$"
    }
```

which should render as:

An example

http://example.com/schemas/example.json	
This is just a tiny example of a schema rendered by sphinx-jsonschema. Yes that's right you can use *reStructuredText* in a description.	
type	*string*
maxLength	100
minLength	10
pattern	^[A-Z]+$

It is also possible to render just a part of an embedded schema using a json pointer (per request Issue 17:

```
.. jsonschema:: #/date

    {
        "title" : "supertitle1",
        "type": "object",
        "properties": {
            "startdate": {"$ref": "#/date"},
            "enddate": {"$ref": "#/date"},
            "manualdate_to1": {"$ref" : "#/manualdate"},
            "definitions1": {"$ref" : "#/definitions/bind"},
            "definitions3": {"$ref" : "#/locbind"}
        },
        "date": {
            "title": "Date",
            "$$target": ["#/date"],
            "description": "YYYY-MM-DD",
            "type": "string"
        }
    }
```

which renders:

Date

YYYY-MM-DD	
type	*string*

1.2.2 Options

There a couple of options implemented in **sphinx-jsonschema** that control the way a schema is rendered or processed. These options are:

lift_title (default: True) Uses the title to create a new section in your document and creates an anchor you can refer to using jsonschema's `$ref` or ReStructuredText's `:ref:` notation. When *False* the title becomes part of the table rendered from the schema, the table cannot be referenced and the option `:lift_description:` is ignored.

lift_description (default: False) Places the description between the title and the table rendering the schema. This option is ignored when `:lift_title:` is *False*.

lift_definitions (default: False) Removed the items under the `definitions` key and renders each of them separately as if they are top-level schemas.

auto_target (default: False) Automatically generate values for the `$$target` key. Especially useful in combination with `:lift_definitions:`.

auto_reference (default: False) Automatically resolves references when possible. Works well with `:auto_target:` and `:lift_definitions:`.

hide_key: (default: None) Hide parts of the schema matching comma separated list of JSON pointers

hide_key_if_empty: (default: None) Hide parts of the schema matching comma separated list of JSON pointers if the value is empty

encoding (default: None) Allows you to define the encoding used by the file containing the json schema.

Lift Title

By default the schema's top level title is displayed above the table containing the remainder of the schema. This title becomes a section that can be included in the table of contents and the index. It is also used to resolve references to the schema from either other schemas of from elsewhere in the documentation.

This option mainly exists to suppress this behaviour. One place where this is desirable is when using jsonschema to validate and document function parameters. See issue 48 for an example.

Lift Description

Lifts the `description` from the table and places it between the title and the table. You will need to have a title defined and the flag **:lift_description:** otherwise it will be included into the table:

which renders:

Example Separate Description

This is just a tiny example of a schema rendered by sphinx-jsonschema.

Whereby the description can shown as text outside the table, and you can still use *reStructuredText* in a description.

http://example.com/schemas/example.json	
type	*string*
maxLength	100
minLength	10
pattern	^[A-Z]+$

Lift Definitions

To separate the `definitions` from the table you will need to have the flag **:lift_definitions:** included. For each item inside the `definitions` it will make a new section with title and a table of the items inside. It's advised to also use the **:auto_reference:** flag to auto link `$ref` to a local `definitions` title.

```
.. jsonschema::
    :lift_definitions:

    {
        "title": "Example with definitions",
        "definitions": {
            "football_player": {
                "type": "object",
                "required": ["first_name", "last_name", "age"],
                "properties": {
                    "first_name": {"type": "string"},
                    "last_name": {"type": "string"},
                    "age": {"type": "integer"}
                }
            },
            "football_team": {
                "type": "object",
                "required": ["team", "league"],
                "properties": {
```

(continues on next page)

(continued from previous page)

```
                    "team": {"type": "string"},
                    "league": {"type": "string"},
                    "year_founded": {"type": "integer"}
                }
            }
        }
    }
```

which renders:

Example with definitions

football_player

type	*object*	
properties		
• **first_name**	type	*string*
• **last_name**	type	*string*
• **age**	type	*integer*

football_team

type	*object*	
properties		
• **team**	type	*string*
• **league**	type	*string*
• year_founded	type	*integer*

Auto Target and Reference

With the **:auto_target:** flag there will be a target created with filename and optional pointer. When you would include auto target on multiple JSON schemas with identical file names it will cause a conflict within your build only the last build target will be used by the references. This also applies if you would embed the schema directly into your documentation; in that case the document name is used as the file name.

With the **:auto_reference:** flag there will be more logic applied to reduce the amount of undefined label warnings. It will check if it is referencing to itself and if there would be a title to link to, when there are titles in the same page that have an identical name it will cause linking issues. If you didn't separate definitions from the schema the `$ref`

will become a text field without a linked reference. If the `$ref` would point to an other schema from the path it will extract the filename it expected to be included into your documentation with **:auto_target:**.

Mainly the **:auto_reference:** flag influences behavior of the existing `$$target` method and could potentially break links.

See below the schema whereby both options are included. For each section it will create a target in this example filename of the document as the schema is added as context and it's pointer if there would be one.
Example of Target & Reference this link as raw text using reStructuredText format would be: **:ref:'directive.rst'**.
And for the definition *person* the raw text would be: **:ref:'directive.rst#/definitions/person'**.

```
.. jsonschema::
    :lift_definitions:
    :auto_reference:
    :auto_target:

{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "title":  "Example of Target & Reference",
    "type": "object",
    "properties": {
        "person": { "$ref": "#/definitions/person" }
    },
    "definitions": {
        "person": {
            "type": "object",
            "properties": {
                "name": { "type": "string" },
                "children": {
                    "type": "array",
                    "items": { "$ref": "#/definitions/person" },
                    "default": []
                }
            }
        }
    }
}
```

which renders:

Example of Target & Reference

type	*object*
properties	
• person	*person*

person

type	*object*	
properties		
• name	type	*string*
• children	type	*array*
	default	
	items	*person*

Setting default values

When you want to use the options :lift_definitions: :lift_description, :auto_target and :auto_reference in most schema renderings it is more convenient to set them once for your whole project.

The `conf.py` option **jsonschema_options** lets you do so. It takes a dict as value the boolean valued keys of which have the same name as the options.

So, in `conf.py` you can state: .. code-block:: py

> **jsonschema_options = {** 'lift_description': True, 'aut_reference': True
>
> }

By default all four options are False.

Overruling defaults

The default values for the options can be overruled by setting the directive options. They accept an optional argument which can be one of the words `On`, `Off`, `True` or `False`. The default value for the argument is `True`.

Declare file encoding

The `:encoding:` option allows you to define the encoding used by the file containing the json schema. When the operating system default encoding does not produce correct results then this option allows you to specify the encoding to use. When omitted the operating system default is used as it always has been. But it is now possible to explicitly declare the expected encoding using `:encoding: utf8`. You can use any encoding defined by Python's codecs for your platform.

Hiding parts of the schema

Sometimes we want to omit certain keys from rendering to make the table more succicnt. This can be achieved using the `:hide_key:` and `:hide_key_if_empty:` options to hide all matching keys or all matching keys with empty associated value, respectively. The options accept comma separated list of JSON pointers. Matching multiple keys is possible using the wildcard syntax `*` for single level matching and `**` for deep matching.

```
.. jsonschema::
    :hide_key: /**/examples
```

This example will hide all `examples` fields regardless of where they are located in the schema. If your JSON pointer contains comma you need to place it inside quotes:

```
.. jsonschema::
    :hide_key: /**/examples,"/**/with, comma"
```

It is also possible to hide a key if their value is empty using `:hide_key_if_empty:`.

```
.. jsonschema::
    :hide_key_if_empty: /**/defaults
```

Prevent escaping of strings

Strings are sometimes subject to multiple evaluation passes when rendering. This happens because *sphinx-jsonschema* renders a schema by transforming in into a table and then recursively call on Sphinx to render the table. To prevent unintended modifications due to this second pass some characters (such as '_' and '*' are escaped before the second pass.

Sometimes that doesn't work out well and you don't want to escape those characters. The option `:pass_unmodified:` accepts one or more JSON pointers and prevents the strings pointed at to be escaped.

```
.. jsonschema::
    :pass_unmodified: /examples/0

    {
        "examples": [
            "unescaped under_score",
            "escaped under_score"
        ]
    }
```

1.3 Schema extensions

1.3.1 $$description

The standard defines the `description` key as having a string value. Since the JSON file format has no provision for some form of line continuation this can result in unwieldy long strings.

To remedy the `$$description` key is introduced. It can be used with and just like the `description` key. It accepts an array of strings which it combines into a single string which is then processed just like the `description`.

This makes it possible to create something like:

```
{
    ...
    "description": "The usual single string description",
    "$$description": [
        "+------------+------------+-----------+",
        "| Header 1   | Header 2   | Header 3  |",
        "+============+============+===========+",
        "| body row 1 | column 2   | column 3  |",
        "+------------+------------+-----------+",
        "| body row 2 | Cells may span columns.|",
        "+------------+------------+-----------+",
        "| body row 3 | Cells may  | - Cells   |",
        "+------------+ span rows. | - contain |",
```

(continues on next page)

(continued from previous page)

```
        "| body row 4 |             | - blocks. |",
        "+------------+-------------+-----------+"
    ],
    ...
}
```

1.3.2 $$target

After some experimentation I concluded that I needed to extend JSON Schema. Most of the time sphinx-jsonschema just does the 'sensible' thing.

The `$ref` key in JSON Schema posed a problem. It works in conjunction with the `id` keyword to implement a schema inclusion method.

I wanted to replace the schema inclusion with a hypertext link to the included schema. Working on a number of large schemas I wanted to document the subschemas as type definitions that are being referenced or used by the main schemas. Therefore I wanted to be able to display the subschema on a different documentation page and have the referring document display a clickable link.

In order to implement this I needed to add the **$$target** key to JSON Schema. `$$target` takes either a single string or an array of strings as parameter.

The string parameter must match the `$ref` parameter **exactly**. So if you are using somewhere the schema:

```
{
    ...
    "$ref": "#/definitions/sample",
    ...
}
```

then the definitions section should read:

```
{
    ...
    "definitions": {
        "sample": {
            "title": "A sample",
            "$$target": "#/definitions/sample"
            ...
        }
    }
}
```

Note: that `$ref` and `$$target` share exactly the same string.

Note: also note the `title` field in `sample`. This is required for the reference to work correctly.

When a referenced schema is used from more than one file it is possible that the value of the `$ref` keywords is not equal.

Consider the case where `schemas/service1/sample.json` and `schemas/service2/sample.json` both reference a `something` subschema located in `schemas/service1/referenced.json` the objects may look like this in schemas/service1/sample.json:

```
{
    ...
    "id": "schemas/service1/sample.json",
    "$ref": "referenced.json#/something",
    ...
}
```

schemas/service2/sample.json would look like:

```
{
    ...
    "id": "schemas/service2/sample.json",
    "$ref": "../service1/referenced.json#/something",
    ...
}
```

This is why `$target` is allowed to have an array of strings as value in referenced.json:

```
{
    ...
    "title": "Something",
    "$$target": ["referenced.json#/something", "../service1/referenced.json#/something
↪"],
    ...
}
```

Combining *$$target*, *$ref* and documentation files

In order to have *$ref* entries be displayed as clickable links you need to:

1. give the referenced schema a *title*,
2. give the referenced schema a *$$target*,
3. include the referenced schema in the documentation.

The title is needed to create a proper section header for the referenced schema. This section header is used to resolve the link generated by the *$ref* key. The title is the link label.

The *$$target* is needed because **sphinx-jsonschema** does not resolve *$ref* like a validator using the *id* key etc. The value of *$$target* should match the corresponding *$ref* value exactly. When the schema is referenced from multiple locations using different values for *$ref* then the value of *$$target* may be an array of strings instead of a single string.

Finally, the referencing and referenced schemas must **both** be included explicitly in the documentation. The referenced schema, when part of a larger schema or set of schemas, can be included using json pointer notation.

Example

The file schema.json contains:

```
{
    "calls": {
        "title": "Allows commercial calls",
        "description": "Person consents to receive commercial offers.",
        "type": "object",
        "properties": {
```

(continues on next page)

(continued from previous page)

```
            "name": {"$ref": "types.json#/Name"},
            "telno": {"$ref": "types.json#/TelephoneNumber"},
            "may_call": {"$ref": "#/definitions/Options"}
        }
    },
    "definitions": {
        "Options": {
            "title": "Options",
            "description": "Embedded definition of type Options",
            "$$target": "#/definitions/Options",
            "type": "string",
            "enum": ["Yes", "No", "Maybe", "Don't care"]
        }
    }
}
```

The file types.json contains:

```
{
    "Name": {
        "title": "Name",
        "description": "Someone's first and lastname",
        "$$target": "types.json#/Name",
        "type": "string",
        "maxLength": 80,
    },
    "TelephoneNumber": {
        "title": "Telephone number",
        "description": "Someone's telephone number",
        "$$target": "types.json#/TelephoneNumber",
        "type": "string",
        "pattern": "[0-9]*"
    }
}
```

The Sphinx source file contains:

```
Caption
#######

Some blahblah about calling people.

.. jsonschema:: schema.json#/calls

More explanations ...

.. jsonschema:: schema.json#/definitions/Options

Types
~~~~~

Introduction on types.

.. jsonschema:: types.json#/Name

More info on Name.
```

(continues on next page)

(continued from previous page)

```
.. jsonschema:: types.json#/TelephoneNumber

Story about TelephoneNumber construction.
```

Which would render as:

Caption

Some blahblah about calling people.

Benefits

This method lets you arrange the schema parts to match the structure of your documentation and also allows you to create multiple copies of a schema in your documentation.

Allows commercial calls

Person consents to receive commercial offers.	
type	*object*
properties	
• name	*Name*
• telno	*Telephone number*
• may_call	*Options*

More explanations . . .

Options

Embedded definition of type Options	
type	*string*
enum	Yes, No, Maybe, Don't care

Types

Introduction on types.

Name

Someone's first and lastname	
type	*string*
maxLength	80

More info on Name.

Telephone number

Someone's telephone number	
type	*string*
pattern	[0-9]*

Story about TelephoneNumber construction.

1.4 Organization

As stated earlier, I needed this to manage and document rather large schemas. I wanted to organize these schemas in such a way that the number of levels remained under control.

To achieve this I wanted the schemas to be able to reference other (reusable) schemas using the `$ref` keyword. These subschemas, should be documented somewhere else but should be all in a single file for performance reasons.

So in order to separate storage and representation I require each `$ref`-erenced subschema to be included explicitly in your .rst file.

1.5 Extending jsonschema

I didn't create jsonschema with extensibility in mind. But I also never thought so many people would find it useful.

1.5.1 Render custom keywords

That being said ankostis needed a way to render his own custom keywords. This is his solution, you need to append this code to your *conf.py* file.

```
## PATCH `sphinx-jsonschema`
#  to render the extra ``units`` and ``tags`` schema properties
#
def _patched_sphinx_jsonschema_simpletype(self, schema):
    """Render the *extra* ``units`` and ``tags`` schema properties for every object."
↪"
    rows = _original_sphinx_jsonschema_simpletype(self, schema)

    if "units" in schema:
        units = schema["units"]
        units = f"``{units}``"
        rows.append(self._line(self._cell("units"), self._cell(units)))
```

(continues on next page)

(continued from previous page)

```
        del schema["units"]

    if "tags" in schema:
        tags = ", ".join(f"``{tag}``" for tag in schema["tags"])
        rows.append(self._line(self._cell("tags"), self._cell(tags)))
        del schema["tags"]

    return rows

sjs_wide_format = importlib.import_module("sphinx-jsonschema.wide_format")
_original_sphinx_jsonschema_simpletype = sjs_wide_format.WideFormat._simpletype  #␣
↪type: ignore
sjs_wide_format.WideFormat._simpletype = _patched_sphinx_jsonschema_simpletype  #␣
↪type: ignore
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Changelog

3.1 Version 1.18.0

Expanding on the work of Pavel Odvody with JSON Pointer the `:pass_unmodified:` option is included. This option prevents escaping the string pointed at.

3.2 Version 1.17.2

Ezequiel Orbe found, reported and fixed a bug escaping backspaces.

3.3 Version 1.17.0

Pavel Odvody contributed the `:hide_key:` directive option. This option allows you to hide certain keys, specified by a JSON Pointer specification, to be excluded from rendering.

3.4 Version 1.16.11

Removed debugging code left in, pointed out by *Kevin Landreth <https://github.com/CrackerJackMack>*.

3.5 Version 1.16.10

iamdbychkov added the `:encoding:` directive option. This option allows explicit control of the encoding used to read a file instead of relying on the operating system default.

3.6 Version 1.16.9

Bugfix.

3.7 Version 1.16.8

Jens Nielsen improved rendering of string values.

3.8 Version 1.16.5-6

Bugfix version.

3.9 Version 1.16.4

Introduces the `:lift_title:` directive option suggested by ankostis. Ankostis also provided an example on how to extend the formatter to handle custom properties.

Fixed a bug in rendering the `items` attribute of the `array` type reported by nijel.

3.10 Version 1.16.1-3

Fixed bugs rendering the `default` and `examples` keywords.

Introduced the configuration entry `jsonschema_options` setting default values for the directive options introduced in 1.16. The options now can accept a parameter to explicitly turn the option on or off.

3.11 Version 1.16

WouterTuinstra reimplemented support for `dependencies` and properly this time. He also improved error handling and reporting and added a couple of options improving the handling of references.

The most important additions are the directive options `:lift_description:`, `:lift_definitions:`, `:auto_target:` and `:auto_reference:`.

In addition to all that he also implemented support for the `if`, `then` and `else` keywords.

3.12 Version 1.15

Add support for the `dependencies` key.

3.13 Versions 1.12, 1.13 and 1.14

Solved several minor bugs.

3.14 Version 1.11

Solved a divergence of the standard reported by bbasic.

3.15 Version 1.10

Ivan Vysotskyy contributed the idea to use an array with the `description` key resulting in the new `$$description` key.

3.16 Version 1.9

Tom Walter contributed the `example` support.

3.17 Version 1.4

Chris Holdgraf contributed Python3 and yaml support.

3.18 Version 1.3

Add unicode support.

3.19 Version 1.2

Improved formatting.

3.20 Version 1.1

Implemented schema cross referencing.

3.21 Version 1.0

Initial release of a functioning plugin.