Django JSON Schema Field Validator Documentation

Release 0.3.0

Chris Lawlor

Jun 15, 2022

CONTENTS:

CHAPTER

ONE

INSTALLATION

1.1 Stable release

To install Django JSON Schema Field Validator, run this command in your terminal:

```
$ pip install jsonfield-validation
```

This is the preferred method to install Django JSON Schema Field Validator, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

CHAPTER

TWO

USAGE

2.1 Validating JSONField

Create a validator by passing a valid JSON schema to `JsonSchemaValidator`:

```
from jsonfield_validation import JsonSchemaValidator

max2 = JsonSchemaValidator({"maxItems": 2})

class MyModel(models.Model):
    items = models.JSONField(validators=[max2])
```

As with any Django model validators, be sure to call `clean_fields` if necessary:

```
instance = MyModel(items=[1, 2, 3])

instance.clean_fields()

django.core.exceptions.ValidationError: {'items': ["[1, 2, 3] is too long"]}
```

2.1.1 JSONField support

`JsonSchemaValidator` is tested against the JSONField implementation included in Django > 3.1, and against django-jsonfield for prior versions of Django.

2.2 Generating JSON schemas with Pydantic

Pydantic is a great fit for Django JSON Field Validator:

```
from django.db import models
from pydantic import BaseModel


class Point(BaseModel):
    x: int
    y: int
```

(continues on next page)

(continued from previous page)

```
class Points(BaseModel):
    __root__: List[Point]


class Shape(models.Model):
    points: models.JSONField(
        validators=[JsonSchemaValidator(schema=Points.schema())]
    )
```

2.3 Testing a schema against existing data

If you're adding schema validation to a model with existing records, you may wish to verify that the existing data will match the proposed schema.

The `JsonSchemaValidator` provides a `check` method, which will run schema validation without raising an exception. This is ideal for validating many objects without the overhead of exception handling.

The `check` method will return an error dict if validation fails, and `None` if it succeeds. Errors are keyed by the flattened path to the errant value. Nested keys are concatenated with a `.` by default. If the error occurs in a list, the errant item's position is noted with square brackets, using standard 0-based indexing.

As an example, here's a nested object containing a list that is meant to be all numbers, but a string snuck in:

```
data = {
    "students": {
        "Alice": {
            "scores": [85, 92, "A"]
        {
    }
}
```

The key in the error dict for this would be `"students.Alice.scores.[2]"`:

```
validator = JsonSchemaValidator(...)

validator.check(data)

{"students.Alice.scores.[2]": "'A' is not a number"}
```

A simple check against all records could then be performed like:

```
validator = JsonSchemaValidator({"maxItems": 2})

if any(validator.check(obj.json_field) for obj in MyModel.objects.iterator()):
    print("Validation failed")
```

Of course, if validation does fail, you'll probably want to know which object failed, and why. A more robust example:

```
validator = JsonSchemaValidator({"maxItems": 2})
error_map = {}
for obj in MyModel.objects.iterator():
    errors = validator.check(obj)
```

(continues on next page)

(continued from previous page)

```
    if errors:
        error_map[obj.id] = errors
```

CHAPTER

THREE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

3.1 Types of Contributions

3.1.1 Report Bugs

Report bugs at https://github.com/chrislawlor/jsonfield-validation/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

3.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

3.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

3.1.4 Write Documentation

Django JSON Schema Field Validator could always use more documentation, whether as part of the official Django JSON Schema Field Validator docs, in docstrings, or even on the web in blog posts, articles, and such.

3.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/chrislawlor/jsonfield_validation/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.2 Get Started!

Ready to contribute? Here's how to set up *jsonfield_validation* for local development.

1. Fork the *jsonfield_validation* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/jsonfield_validation.git
```

3. If you're using asdf, run `asdf install` to install the Pythons listed in `.tool-versions`. Add the asdf-python plugin if you don't have it already.
4. Install your local copy into a virtualenv. Assuming you have tox installed, this is how you set up your fork for local development:

```
$ make develop
```

   This will create a virtual environment in the `.env` directory, enable it, and install all project and development dependencies.

   Alternately, with virtualenvwrapper:

   $ mkvirtualenv jsonfield_validation $ cd jsonfield_validation/ $ python setup.py develop $ python -m pip install -r requirements_dev.txt

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ make lint
$ make test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv. Check `tox.ini` for the Python versions you'll need available. If you're using asdf, simply run `asdf install`.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

3.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6, 3.7, 3.8, and 3.9. Check https://travis-ci.com/chrislawlor/jsonfield_validation/pull_requests and make sure that the tests pass for all supported Python versions.

3.4 Tips

To run a subset of tests:

```
$ pytest tests.test_jsonfield_validation
```

3.5 Publishing

A reminder for the maintainers on how to publish. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER

FOUR

HISTORY

4.1 0.3.0 (2022-06-15)

- Add path to invalid value to error_list message,

4.2 0.2.0 (2022-06-15)

- Add check method.

4.3 0.1.0 (2022-06-14)

- First release on PyPI.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search