Flask Sieve

Release 1.1.2

Edward Njoroge

Sep 13, 2020

CONTENTS

This package provides an approach to validating incoming requests using powerful and composable rules.

CHAPTER

ONE

INSTALLATION

To install and update using pip.

```
pip install -U flask-sieve
```

CHAPTER

TWO

QUICKSTART

To learn about these powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user.

2.1 Auto-validation of Requests

Suppose you had a simple application with an endpoint to register a user.

```
flask-app/
  __init__.py
  app.py
  app_requests.py
```

We are going to create validations for this endpoint.

```
# app.py

from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/', methods=('POST',))
def register():
    return jsonify({'message': 'Registered!'}), 200

app.run()
```

To validate incoming requests to this endpoint, we create a class with validation rules of registering a user as follows:

```
# app_requests.py

from flask_sieve import FormRequest

class RegisterRequest(FormRequest):
    def rules(self):
        return {
            'email': ['required', 'email'],
            'username': ['required', 'string', 'min:6'],
            'password': ['required', 'min:6', 'confirmed']
        }
```

Now, using this class, we can guard our endpoint using a `validate` decorator.

```
# app.py

from flask import Flask
from flask_sieve import Sieve, validate
from .app_requests import RegisterRequest

app = Flask(__name__)
Sieve(app)

@app.route('/', methods=('POST',))
@validate(RegisterRequest)
def register():
    return jsonify({'message': 'Registered!'}), 200

app.run()
```

Note the initialization of `Sieve` with the application instance. This is required for setting up the necessary mechanism to autorespond with the error messages.

2.2 Manual Validation of Requests

Sometimes you might not wish to rely on the default auto-response made by Flask-Sieve. In this case, you can create an instance of `Validator` and set the rules yourself.

Using the same application shown above, this is how you would go about it:

```
# app.py

from flask import Flask, jsonify, request
from flask_sieve import Sieve, Validator

app = Flask(__name__)
Sieve(app)

@app.route('/', methods=('POST',))
def register():
    rules = {
        'email': ['required', 'email'],
        'avatar': ['image', 'dimensions:200x200'],
        'username': ['required', 'string', 'min:6'],
    }
    validator = Validator(rules=rules, request=request)
    if validator.passes():
        return jsonify({'message': 'Registered!'}), 200
    return jsonify(validator.messages()), 400

app.run()
```

This would allow you to make the correct response in cases where you would not want to rely on the response format provided by Flask-Sieve.

CHAPTER

THREE

DIGGING DEEPER

Flask-Sieve supports various approaches to validating requests. Here we will make an in-depth tour of the functionalities offered.

3.1 Form vs JSON Requests

To address the differences in requests with form requests (with `Content-Type: 'multipart/form-data'`) and JSON requests (with `Content-Type: 'application/json'`) Flask-Sieve supports two kinds of auto-validating requests:

3.1.1 Form Requests

To validate form requests, you have to inherit from `FormRequest` on the validation request. Example:

```
from flask_sieve import FormRequest

class PostRequest(FormRequest):
    def rules(self):
        return {
            'image': ['file'],
            'username': ['required', 'string', 'min:6'],
        }
```

3.1.2 JSON Requests

To validate this format you will have to inherit from `JsonRequest`. Before validating the request, this checks that the request is indeed a JSON request (with `Content-Type: 'application/json'`).

```
from flask_sieve import JsonRequest

class PostRequest(JsonRequest):
    def rules(self):
        return {
            'email': ['required', 'email'],
        }
```

3.2 Error Messages Format

In case validation fails to pass, the following is the format of the generated response:

```
{
    success: False,
    message: 'Validation error',
    errors: {
        'email': [
            'The email is required.',
            'The email must be a valid email address.'
        ],
        'password': [
            'The password confirmation does not match.'
        ]
    }
}
```

All validation error messages will have a HTTP error status code 400.

3.3 Stopping on First Validation Failure

Sometimes you may wish to stop running validation rules on an attribute after the first validation failure. To do so, assign the `bail` rule to the attribute:

```
# app_requests.py

# ... omitted for brevity ...
def rules(self):
    return {
        'body': ['required'],
        'title': ['bail', 'string', 'required', 'max:255'],
    }
```

In this example, if the `string` rule on the `title` attribute fails, the `max` rule will not be checked. Rules will be validated in the order they are assigned.

3.4 A Note on Nested Attributes

If your HTTP request contains "nested" parameters, you may specify them in your validation rules using "dot" syntax:

```
# app_requests.py

# ... omitted for brevity ...

def rules(self):
    return {
        'author.name': ['required'],
        'author.description': ['required'],
    }
```

3.5 Customizing the Error Messages

You may customize the error messages used by the form request by overriding the `messages` method. This method should return an array of attribute / rule pairs and their corresponding error messages:

```
# app_requests.py

from flask_sieve import FormRequest

class RegisterRequest(FormRequest):
    def messages(self):
        return {
            'email.required': 'The email is required',
            'password.confirmed': 'Password must be at least 6 characters'
        }

    def rules(self):
        return {
            'email': ['required', 'email'],
            'username': ['required', 'string', 'min:6'],
            'password': ['required', 'min:6', 'confirmed',]
        }
```

3.6 Adding Custom Rules

Besides the rules offered by default, you can extend the validator with your own custom rules. You can do this either when defining the Form/JSON Request class or when you instantiate a `Validator`.

3.6.1 Custom Rule Handler

A rule handler is a predicate (a method returning either `True` or `False`) that you can use to add your validations to Flask-Sieve.

This method must satisfy the following conditions:

- It must start with the `validate_` keyword.
- It will receive keyword the following keyword parameters:
  - `value` - the value of the request field being validated.
  - `attribute` - the field name being validated.
  - `params` - a list of parameters passed to the rule. For instance, for the inbuilt rule `between:min,max`, the list will be `[min, max]`.
  - `nullable` - a boolean marking whether this field has been specified as `nullable` or not.
  - `rules` - a list containing all the rules passed on the field.

Tip: In case your handler does not need all these parameters, you can simply ignore the ones you don't need with `**kwargs`.

For example:

```
def validate_odd(value, **kwargs):
    return int(value) % 2
```

3.6.2 Custom Rules on Form/JSON Requests

To define a custom rule validator on a Form/JSON Rquest, you will have to provide it in a method named `custom_handlers` as follows:

```
from flask_sieve import FormRequest

def validate_odd(value, **kwargs):
    return int(value) % 2

class RegisterRequest(FormRequest):

    # ... omitted for brevity ...

    def custom_handlers(self):
        return [{
            'handler': validate_odd,  # the rule handler
            'message': 'Number must be odd', # the message to display when this rule␣
↪fails
            'params_count': 0 # the number of parameters the rule expects
        }]
```

3.6.3 Custom Rules on `Validator` Instance

To add a custom rule handler to a `Validator` instance, you have will have to use `register_rule_handler` method as shown below:

```
from flask import Flask, jsonify, request
from flask_sieve import Sieve, Validator

app = Flask(__name__)
Sieve(app)

def validate_odd(value, **kwargs):
    return int(value) % 2

@app.route('/', methods=('POST',))
def register():
    rules = {'avatar': ['image', 'dimensions:200x200']}
    validator = Validator(rules=rules, request=request)
    validator.register_rule_handler(
        handler=validate_odd,
        message='Must be odd',
        params_count=0
    )
    if validator.passes():
        return jsonify({'message': 'Registered!'}), 200
    return jsonify(validator.messages()), 400
```

CHAPTER

FOUR

AVAILABLE VALIDATIONS

4.1 accepted

The field under validation must be *yes*, *on*, *1*, or *true*. This is useful for validating "Terms of Service" acceptance.

4.2 active_url

The field under validation must be active and responds to a request from `requests` Python package.

4.3 after:*date*

The field under validation must be a value after a given date. The dates will be passed into the `parse` function from `python-dateutil` Python

```
'start_date': ['required', 'date', 'after:2018-02-10']
```

4.4 after_or_equal:*date*

The field under validation must be a value after or equal to the given date.

4.5 alpha

The field under validation must be entirely alphabetic characters.

4.6 alpha_dash

The field under validation may have alpha-numeric characters, as well as dashes and underscores.

4.7 alpha_num

The field under validation must be entirely alpha-numeric characters.

4.8 array

The field under validation must be an `array` string.

4.9 bail

Stop running validation rules after the first validation failure.

4.10 before:*date*

The field under validation must be a value preceding the given date. The dates will be passed into the Python `python-dateutil` package.

4.11 before_or_equal:*date*

The field under validation must be a value preceding or equal to the given date. The dates will be passed into the `parse` function from `python-dateutil` Python package.

4.12 between:*min*,*max*

The field under validation must have a size between the given *min* and *max*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

4.13 boolean

The field under validation must be able to be cast as a boolean. Accepted input are `true`, `false`, `1`, `0`, `"1"`, and `"0"`.

4.14 confirmed

The field under validation must have a matching field of `foo_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

4.15 date

The field under validation must be a valid, non-relative date according to the `parse` function of `python-dateutil`.

4.16 date_equals:*date*

The field under validation must be equal to the given date. The dates will be passed into the `parse` function of python-dateutil.

4.17 different:*field*

The field under validation must have a different value than *field*.

4.18 digits:*value*

The field under validation must be *numeric* and must have an exact length of *value*.

4.19 digits_between:*min*,*max*

The field under validation must have a length between the given *min* and *max*.

4.20 dimensions

The file under validation must be an image meeting the dimension constraints specified as `WidthxHeight`

```
'avatar': ['dimensions:200x200']
```

4.21 distinct

When working with arrays, the field under validation must not have any duplicate values.

```
'foo': ['distinct']
```

4.22 email

The field under validation must be formatted as an e-mail address.

4.23 file

The field under validation must be a successfully uploaded file.

4.24 filled

The field under validation must not be empty when it is present.

4.25 gt:*field*

The field under validation must be greater than the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

4.26 gte:*field*

The field under validation must be greater than or equal to the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

4.27 image

The file under validation must be an image (jpeg, png, bmp, gif, tif, or svg)

4.28 in:*foo,bar,...*

The field under validation must be included in the given list of values.

4.29 in_array:*anotherfield*

The field under validation must exist in *anotherfield*'s values.

4.30 integer

The field under validation must be an integer.

4.31 ip

The field under validation must be an IP address.

4.32 ipv4

The field under validation must be an IPv4 address.

4.33 ipv6

The field under validation must be an IPv6 address.

4.34 json

The field under validation must be a valid JSON string.

4.35 lt:*field*

The field under validation must be less than the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

4.36 lte:*field*

The field under validation must be less than or equal to the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

4.37 max:*value*

The field under validation must be less than or equal to a maximum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

4.38 mime_types:*text/plain,...*

The file under validation must match one of the given MIME types:

```
'video': ['mime_types:video/avi,video/mpeg,video/quicktime']
```

To determine the MIME type of the uploaded file, the file's contents will be read and the framework will attempt to guess the MIME type, which may be different from the client provided MIME type.

4.39 min:*value*

The field under validation must have a minimum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

4.40 not_in:*foo,bar,...*

The field under validation must not be included in the given list of values.

4.41 not_regex:*pattern*

The field under validation must not match the given regular expression.

4.42 nullable

The field under validation may be `None`. This is particularly useful when validating primitive such as strings and integers that can contain `None` values.

4.43 numeric

The field under validation must be numeric.

4.44 present

The field under validation must be present in the input data but can be empty.

4.45 regex:*pattern*

The field under validation must match the given regular expression.

4.46 required

The field under validation must be present in the input data and not empty. A field is considered "empty" if one of the following conditions are true:

- The value is `None`.
- The value is an empty string.
- The value is an empty array.

4.47 required_if:*anotherfield*,*value*,...

The field under validation must be present and not empty if the *anotherfield* field is equal to any *value*.

4.48 required_unless:*anotherfield*,*value*,...

The field under validation must be present and not empty unless the *anotherfield* field is equal to any *value*.

4.49 required_with:*foo*,*bar*,...

The field under validation must be present and not empty *only if* any of the other specified fields are present.

4.50 required_with_all:*foo*,*bar*,...

The field under validation must be present and not empty *only if* all of the other specified fields are present.

4.51 required_without:*foo*,*bar*,...

The field under validation must be present and not empty *only when* any of the other specified fields are not present.

4.52 required_without_all:*foo*,*bar*,...

The field under validation must be present and not empty *only when* all of the other specified fields are not present.

4.53 same:*field*

The given *field* must match the field under validation.

4.54 size:*value*

The field under validation must have a size matching the given *value*. For string data, *value* corresponds to the number of characters. For numeric data, *value* corresponds to a given integer value. For an array, *size* corresponds to the `count` of the array. For files, *size* corresponds to the file size in kilobytes.

4.55 starts_with:*foo*,*bar*,...

The field under validation must start with one of the given values.

4.56 string

The field under validation must be a string. If you would like to allow the field to also be `None`, you should assign the `nullable` rule to the field.

4.57 timezone

The field under validation must be a valid timezone identifier according to the `pytz` Python package.

4.58 url

The field under validation must be a valid URL.

4.59 uuid

The field under validation must be a valid RFC 4122 (version 1, 3, 4, or 5) universally unique identifier (UUID).

CHAPTER

FIVE

CONTRIBUTIONS

Contributions and bugfixes are welcome!

CHAPTER

SIX

LICENSE (BSD-2)

A Flask package for validating requests (Inspired by Laravel).

Find a copy of the License here.