pre-request Documentation

wudong@eastwu.cn

Jan 26, 2022

CONTENTS

1 Installation **3**
- 1.1 Python Version . . . 3
- 1.2 Dependencies . . . 3
- 1.3 Install pre-request with pip . . . 3

2 Quickstart **5**
- 2.1 Minimal Example . . . 5
- 2.2 *pre* singleton . . . 6
- 2.3 *Flask* Extension . . . 6
- 2.4 Decorator . . . 6
- 2.5 Use parse . . . 7

3 Validate Rules **9**
- 3.1 location . . . 11
- 3.2 deep . . . 11
- 3.3 type . . . 11
- 3.4 skip . . . 11
- 3.5 multi . . . 12
- 3.6 structure . . . 12
- 3.7 required . . . 12
- 3.8 required_with . . . 12
- 3.9 default . . . 12
- 3.10 split . . . 13
- 3.11 trim . . . 13
- 3.12 enum . . . 13
- 3.13 reg . . . 13
- 3.14 contains . . . 13
- 3.15 contains_any . . . 14
- 3.16 excludes . . . 14
- 3.17 startswith . . . 14
- 3.18 not_endswith . . . 14
- 3.19 endswith . . . 14
- 3.20 not_endswith . . . 15
- 3.21 lower . . . 15
- 3.22 upper . . . 15
- 3.23 ipv4/ipv6 . . . 15
- 3.24 mac . . . 15
- 3.25 url_encode . . . 16
- 3.26 url_decode . . . 16
- 3.27 alpha . . . 16

3.28 alphanum . . . 16
3.29 numeric . . . 16
3.30 number . . . 17
3.31 email . . . 17
3.32 fmt . . . 17
3.33 latitude / longitude . . . 17
3.34 eq / eq_key . . . 17
3.35 neq / neq_key . . . 18
3.36 gt / gt_key . . . 18
3.37 gte / gte_key . . . 18
3.38 lt / lt_key . . . 18
3.39 lte / lte_key . . . 18
3.40 dest . . . 19
3.41 json . . . 19
3.42 callback . . . 19

4 Customize **21**
4.1 Response . . . 21
4.2 Formatter . . . 22
4.3 Filter . . . 22
4.4 Store Key . . . 23

Welcome to pre-request's document. This framework is designed to validate params for *Flask* request params, this framework can validate complex struct and field, including Cross Field, Cross Struct.

This part of the documentation will instruct how to use pre-request in your flask project

CHAPTER

ONE

INSTALLATION

1.1 Python Version

We recommend using the latest version of python 3. pre-request supports Python 3.5 and newer and PyPy

1.2 Dependencies

This framework is designed for *Flask*, therefore *Flask* will be installed automatically when installing pre-request

1.3 Install pre-request with pip

You can use the following command to install pre-request:

```
pip install pre-request
```

CHAPTER

TWO

QUICKSTART

Eager to get started? This page gives a good example to use pre-request. It assumes you already have *pre-request* installed If you do not, head over to the Installation section.

2.1 Minimal Example

A minimal example looks something like this:

```
from flask import Flask

from pre_request import pre
from pre_request import Rule

app = Flask(__name__)

req_params = {
   "userId": Rule(type=int, required=True)
}

@app.route("/")
@pre.catch(req_params)
def hello_world(params):
   return str(params)
```

what happened in this code ?

1. Use *pre-request* library to import a global object *pre*
2. Define request params rule, *userId* must be type of *int* and required
3. Use *@pre.catch(req_params)* to filter input value
4. Use *~flask.g* or *def hello_world(params)* to get formatted input value

2.2 *pre* singleton

pre-request support global singleton object, we can use this object to update runtime params

- *pre.fuzzy* pre-request will fuzzy error message to avoid expose sensitive information
- *pre.sore_key* use another params to store formatted request info
- *pre.content_type* pre-request will response html or json error message, use *application/json* or *text/html*
- *pre.skip_filter* pre-request will ignore all of the check filter, but *dest* is still valid.

2.3 *Flask* Extension

pre-request support flask extension configuration to load params.

```
app = Flask(__name__)

app.config["PRE_FUZZY"] = True
app.config["PRE_STORE_KEY"] = "pp"
app.config["PRE_CONTENT_TYPE"] = "application/json"
app.config["PRE_SKIP_FILTER"] = False

pre.init_app(app=app)
```

2.4 Decorator

pre-request use decorator *pre.catch* to validate request params with special kind of method

```
@app.route("/get", methods=['get'])
@pre.catch(get=req_params)
def get_handler(params):
    return str(params)

@app.route("/post", methods=['post'])
@pre.catch(post=req_params)
def get_handler(params):
    return str(params)
```

we can also support validate different rule for different request method.

```
@app.route("/all", methods=['get', 'post'])
@pre.catch(get=get_field, post=post_field)
def all_handler(params):
    return str(params)
```

you can validate params for all of the request methods with no key.

```
@app.route("/all", methods=['get', 'post'])
@pre.catch(rules)
def all_handler(params):
    return str(params)
```

2.5 Use parse

We can use function *pre.parse* instead of decorator *@pre.catch()*. At this mode, you must catch *ParamsValueError* by yourself.

```
args = {
    "params": Rule(email=True)
}

@app.errorhandler(ParamsValueError)
def params_value_error(e):
    return pre.fmt_resp(e)


@app.route("/index")
def example_handler():
    rst = pre.parse(args)
    return str(rst)
```

CHAPTER

THREE

VALIDATE RULES

Control:

Rule	Desc
type	Direct type
required	Param is required
default	Default value if param is empty
dest	Direct key for result
required_with	Required with other key
location	Which location to read value for request
skip	Skip all of the filters

Other:

Rule	Desc
json	Json deserialize value
callback	Custom callback function

Fields:

Rule	Desc
eq_key	Field equal to another field
neq_key	Field not equal to another field
gt_key	Field greater than another field
gte_key	Field greater than or equal to another field
lt_key	Field less than another field
lte_key	Field less than or equal to another field

Network:

Rule	Desc
ipv4	Internet protocol address IPv4
ipv6	Internet protocol address IPv6
mac	Media access control address MAC
url_encode	Url encode with *urllib.parse.quote*
url_decode	Url decode with *urllib.parse.unquote*

Strings:

Rule	Desc
len	Content length for string or array
trim	Trim space characters
reg	Regex expression
contains	Contains
contains_any	Contains any items
excludes	Excludes
startswith	Starts with
not_startswith	Not start with
not_endswith	Not end with
endswith	Ends with
lower	Lowercase
upper	Uppercase
Split	Split string with special character

Format:

Rule	Desc
fmt	*date* or *datetime* format
latitude	Latitude
longitude	Longitude
structure	Describe substructure for array or dict
multi	Value is array
deep	Find value from substructure
enum	Enum value
alpha	Alpha only
alphanum	Alpha or numeric
numeric	Numeric
number	Number only
email	Email address for RFC5322

Comparison:

Rule	Desc
eq	Equals
neq	Not equal
gt	Greater than
gte	Greater than or equal
lt	Less than
lte	Less than or equal

3.1 location

By default, *pre-request* try to parse values form *flask.Request.values* and *flask.Request.json*. Use *location* to specify location to get values. current support [“args”, “form”, “values”, “headers”, “cookies”, “json”]

```
params = {
  "Access-Token": Rule(location="headers"),
  "userId": Rule(location=["cookies", "headers", "args"])
}
```

3.2 deep

By default, *pre-request* can parse value from complex structure. we can use *deep=False* to turn off this feature, pre-request will parse values from top level.

```
params = {
  "userInfo": {
      "userId": Rule(type=int, required=False),
      "socialInfo": {
          "gender": Rule(type=int, enum=[1, 2], default=1),
          "age": Rule(type=int, gte=18, lt=80),
          "country": Rule(required=True, deep=False)
      }
  }
}
```

3.3 type

pre-request try to convert value to special type.

```
params = {
   "userId": Rule(type=int)
}
```

3.4 skip

pre-request will skip validate value at this field. we will put origin value in the result structure.

```
params = {
   "userName": Rule(skip=True)
}
```

3.5 multi

if you set *multi=True*, we will check every items in array. otherwise it will be regarded as a whole

```
params = {
    "userIds": Rule(type=int, multi=True)
}
```

3.6 structure

You can use *structure* field to define sub structure in array. This field will be only valid in *multi=True*.

params = {

"friends": Rule(multi=True, structure={ "userId": Rule(type=int, required=True), "userName": Rule(type=str, required=True)

})

}

3.7 required

pre-request validate the value is not None or user do not input this value. Specially, if user don't input this value and *skip=True*, *pre-request* will fill it with *missing* type.

```
params = {
    "profile": Rule(required=False)
}
```

3.8 required_with

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

```
params = {
     "nickName": Rule(required=False),
     "profile": Rule(required=False, required_with="nickName")
}
```

3.9 default

pre-request will fill the default value into the field only if the field is not required and current value is None

```
params = {
   "nickName": Rule(required=False, default="")
}
```

3.10 split

pre-request will split origin string value with special char and the check rule will filter to every value in the result array

```
params = {
  "userId": Rule(int, split=",")
}
```

3.11 trim

pre-request will try to remove the space characters at the beginning and end of the string.

```
params = {
   "nickName": Rule(trim=True)
}
```

3.12 enum

Ensure that the parameters entered by the user are within the specified specific value range.

```
params = {
   "gender": Rule(direct_type=int, enum=[1, 2])
}
```

3.13 reg

Use regular expressions to verity that the user input string meets the requirements.

```
params = {
   "tradeDate": Rule(reg=r"^[1-9]\d{3}-(0[1-9]|1[0-2])-(0[1-9]|[1-2][0-9]|3[0-1])$")
}
```

3.14 contains

Ensure that the field entered by the user contain all of the special value.

```
params = {
  "content": Rule(contains=["", ""])
}
```

3.15 contains_any

Ensure that the field entered by the user contain any of the special value.

```
params = {
  "content": Rule(contains_any=["", ""])
}
```

3.16 excludes

Ensure that the field entered by the user can not contain any of special value.

```
params = {
   "content": Rule(excludes=["", ""])
}
```

3.17 startswith

Ensure that the input string value must be start with special substring

```
params = {
   "nickName": Rule(startswith="CN")
}
```

3.18 not_endswith

Ensure that the input string value must be not start with special substring

```
params = {
   "nickName": Rule(not_startswith="USA")
}
```

3.19 endswith

Ensure that the input string value must be end with special substring

```
params = {
   "email": Rule(endswith="@eastwu.cn")
}
```

3.20 not_endswith

Ensure that the input string value must be not end with special substring

```
params = {
    "email": Rule(not_endswith="@eastwu.cn")
}
```

3.21 lower

pre-request will convert all characters in the string to lowercase style.

```
params = {
  "nickName": Rule(lower=True)
}
```

3.22 upper

pre-request will convert all characters in the string to uppercase style.

```
params = {
  "country": Rule(upper=True)
}
```

3.23 ipv4/ipv6

Ensure that the field entered by the user conform to the ipv4/6 format.

```
params = {
  "ip4": Rule(ipv4=True)
  "ip6": Rule(ipv6=True)
}
```

3.24 mac

Ensure that the field entered by the user conform to the MAC address format.

```
params = {
  "macAddress": Rule(mac=True)
}
```

3.25 url_encode

Encode url by function *urllib.parse.quote*. This rule is only valid for parameters of type *str*. You can select the encoding type through the *encoding* parameter.

```
params = {
  "url": Rule(type=str, url_encode=True, encoding="GB2312")
}
```

3.26 url_decode

Decode url by function *urllib.parse.unquote*. This rule is only valid for parameters of type *str*. You can select the encoding type through the *encoding* parameter.

```
params = {
  "url": Rule(type=str, url_decode=True, encoding="GB2312")
}
```

3.27 alpha

Check that a string can oly consist of letters.

```
params = {
  "p": Rule(type=str, alpha=True)
}
```

3.28 alphanum

Check that a string can oly consist of letters or numeric.

```
params = {
  "p": Rule(type=str, alphanum=True)
}
```

3.29 numeric

Check that a string can oly consist of numeric.

```
params = {
  "p": Rule(type=str, numeric=True)
}
```

3.30 number

Check that a string can oly consist of number.

```
params = {
  "p": Rule(type=str, number=True)
}
```

3.31 email

Check that a string is valid email address.

```
params = {
  "p": Rule(type=str, email=True)
}
```

3.32 fmt

Provides the style when the string is converted to *datetime* or *date* type. This is valid only on *type=datetime.datetime*

```
params = {
  "birthday": Rule(type=datetime.datetime, fmt="%Y-%m-%d"),
  "otherDate": Rule(type=datetime.date, fmt="%Y-%m-%d")
}
```

3.33 latitude / longitude

Ensure that the field entered by the user conform to the *latitude/longitude* format.

```
params = {
  "latitude": Rule(latitude=True),
  "longitude": Rule(longitude=True)
}
```

3.34 eq / eq_key

Used to check whether the user input parameter is equal to another value or another parameter.

```
params = {
  "userId": Rule(eq=10086),
  "userId2": Rule(eq_key="userId")
}
```

3.35 neq / neq_key

Used to check whether the user input parameter is not equal to another value or another parameter.

```
params = {
    "userId": Rule(neq=0),
    "forbidUserId": Rule(neq_key="userId")
}
```

3.36 gt / gt_key

Used to check whether the user input parameter is great than another value or another parameter.

```
params = {
    "kidAge": Rule(type=int, gt=0),
    "fatherAge": Rule(type=int, gt_key="kidAge")
}
```

3.37 gte / gte_key

Used to check whether the user input parameter is great than or equal to another value or another parameter.

```
params = {
    "kidAge": Rule(type=int, gte=0),
    "brotherAge": Rule(type=int, gte_key="kidAge")
}
```

3.38 lt / lt_key

Used to check whether the user input parameter is less than another value or another parameter.

```
params = {
    "fatherAge": Rule(type=int, lt=100),
    "kidAge": Rule(type=int, lt_key="fatherAge")
}
```

3.39 lte / lte_key

Used to check whether the user input parameter is less than or equal to another value or another parameter.

```
params = {
    "fatherAge": Rule(type=int, lte=100),
    "kidAge": Rule(type=int, lte_key="fatherAge")
}
```

3.40 dest

We will convert the key of the parameter to another value specified.

```
params = {
  "userId": Rule(type=int, dest="user_id")
}
```

3.41 json

We will try to use the *json.loads* method to parse the value of the parameter to convert it into a *list* or *dict* type.

3.42 callback

If the filters we provide cannot meet your needs, you can pass in the parse function you defied through the *callback* method.

```
def hand(value):
  if value <= 10:
      raise ParamsValueError("'userId' must be greater than 10")
  return value + 100

params = {
  "userId": Rule(type=int, callback=hand)
}
```

CHAPTER

FOUR

CUSTOMIZE

4.1 Response

Normally, when pre-request finds that the user input parameter does not meet the requirements, it will directly interrupt the processing and return the discovered problem to the requester. The default JSON type of response format provided by pre-request is as follows:

```
{
    "respCode": 400,
    "respMsg": "Error Message",
    "result": {}
}
```

In some scenarios, we need different response formats. Pre-request provides the ability to customize the response. You need to implement a class that inherits from `BaseResponse` to implement your own data response processing.

```
from flask import make_response
from pre_request import BaseResponse

class CustomResponse(BaseResponse):

  def make_response(
          cls,
          error: "ParamsValueError",
          fuzzy: bool = False,
          formatter: t.Optional[t.Callable] = None
  ):
      result = {
          "code": 900,
          "rst": {}
      }

      from flask import make_response  # pylint: disable=import-outside-toplevel
      response = make_response(json.dumps(result))
      response.headers["Content-Type"] = "application/json; charset=utf-8"
      return response
```

```
from pre_request import pre

pre.add_response(CusResponse)
```

4.2 Formatter

If you feel that the custom response class is too complicated, we also provide the function of a custom formatting function. The pre-request will give priority to calling your custom function to generate a response string.

```
def custom_formatter(error: ParamsValueError):
  """
  """
  return {
      "code": 411,
      "msg": "hello",
      "sss": "tt",
  }
```

```
from pre_request import pre
pre.add_formatter(custom_formatter)
```

4.3 Filter

pre-request pre-requestpre-request pre-request

`BaseFilter`

```
from pre_request import BaseFilter

class CustomFilter(BaseFilter):

    def filter_required(self):
        """
        """
        return True

    def __call__(self, *args, **kwargs):
        """
        """
        super().__call__()

        if self.rule.direct_type == int and self.key == "number" and self.value !=␣
↪10086:
            raise ParamsValueError(message="any error messages you want")

        return self.value + 1
```

filter_required *__call__* *filter_required* *__call__*

pre-request

```
from pre_request import pre

pre.add_filter(CustomFilter)
```

4.4 Store Key

By default, pre-request stores formatted input parameters in *~flask.g.params* and the *params* parameter of the current function You can set the *store_key* parameter of the pre-request to change the storage key of the parameter.

```
from pre_request import pre
pre.store_key = "pre_params"
```