django-vimage Documentation

Release 0.1.0

Nick Mavrakis

Jun 07, 2022

Contents

1 Introduction **1**
- 1.1 Quickstart . . . 1
- 1.2 Features . . . 4
- 1.3 Running Tests . . . 4
- 1.4 Future additions . . . 5
- 1.5 Credits . . . 5

2 Installation **7**

3 Usage **9**
- 3.1 `VIMAGE` key . . . 9
- 3.2 `VIMAGE` value . . . 10

4 Configuration **13**
- 4.1 `'SIZE'` . . . 14
- 4.2 `'DIMENSIONS'` . . . 14
- 4.3 `'FORMAT'` . . . 16
- 4.4 `'ASPECT_RATIO'` . . . 17

5 How it works **19**

6 Examples **21**
- 6.1 With specificity 1 . . . 21
- 6.2 With specificity 2 . . . 22
- 6.3 With specificity 3 . . . 22
- 6.4 With specificity 1 + 2 . . . 23
- 6.5 With specificity 1 + 3 . . . 24

7 Reference **25**
- 7.1 `vimage.core.base` . . . 25
- 7.2 `vimage.core.checker` . . . 25
- 7.3 `vimage.core.const` . . . 25
- 7.4 `vimage.core.validator_types` . . . 25

8 Contributing **27**
- 8.1 Types of Contributions . . . 27
- 8.2 Get Started! . . . 28

8.3 Pull Request Guidelines . . . 29
8.4 Tips . . . 29

9 Credits **31**
9.1 Development Lead . . . 31
9.2 Contributors . . . 31

10 History **33**
10.1 0.1.0 (2018-04-17) . . . 33

11 Indices and tables **35**

Python Module Index **37**

Index **39**

CHAPTER 1

Introduction

Django Image validation for the Django Admin as a breeze. Validations on: Size, Dimensions, Format and Aspect Ratio.

Because, I love to look for the origin of a word/band/place/something, this package name comes from the word *validate* and (you guessed it) *image*. Thus, `django-vimage`. Nothing more, nothing less :)

This package was created due to lack of similar Django packages that do image validation. I searched for this but found nothing. So, I decided to create a reusable Django package that will do image validation in a simple manner. Just declare some `ImageFields` and the rules to apply to them in a simple Python dictionary. Firstly, I wrote the blueprint on a piece of paper and then I, gradually, ported it to Django/Python code.

1.1 Quickstart

Install django-vimage

```
pip install django-vimage
```

Add it to your `INSTALLED_APPS`

```
INSTALLED_APPS = (
    ...
    'vimage.apps.VimageConfig',
    ...
)
```

Finally, add the `VIMAGE` dict configuration somewhere in your `settings` file

```
VIMAGE = {
    'my_app.models': {
        'DIMENSIONS': (200, 200),
        'SIZE': {'lt': 100},
    }
}
```

The above `VIMAGE` setting sets the rules for all Django `ImageField` fields under the `my_app` app. More particular, all `ImageFields` should be 200 x 200px **and** less than 100KB. Any image than violates any of the above rules, a nice-looking error message will be shown (translated accordingly) in the Django admin page.

A full example of possible key:value pairs is shown below. Note that the following code block is not suitable for copy-paste into your `settings` file since it contains duplicate dict keys. It's just for demonstration. For valid examples, refer to the *examples*.

```
 VIMAGE = {
     # Possible keys are:
     # 'app.models'  # to all ImageFields inside this app
     # 'app.models.MyModel'  # to all ImageFields inside MyModel
     # 'app.models.MyModel.field'  # only to this ImageField

     # Example of applying validation rules to all images across
     # all models of myapp app
     'myapp.models': {
         # rules
     },

     # Example of applying validation rules to all images across
     # a specific model
     'myapp.models.MyModel': {
         # rules
     },

     # Example of applying validation rules to a
     # specific ImageField field
     'myapp.models.MyModel.img': {
         # rules
     },

     # RULES
     'myapp.models': {

         # By size (measured in KB)

         # Should equal to 100KB
         'SIZE': 100,  # defaults to eq (==)

         # (100KB <= image_size <= 200KB) AND not equal to 150KB
         'SIZE': {
```

(continues on next page)

(continued from previous page)

```
            'gte': 100,
            'lte': 200,
            'ne': 150,
        },

        # Custom error message
        'SIZE': {
            'gte': 100,
            'lte': 200,
            'err': 'Your own error message instead of the default.'
                   'Supports <strong>html</strong> tags too!',
        },


        # By dimensions (measured in px)
        # Should equal to 1200x700px (width x height)
        'DIMENSIONS': (1200, 700),  # defaults to eq (==)

        # Should equal to one of these sizes 1000x300px or 1500x350px
        'DIMENSIONS': [(1000, 300), (1500, 350)],

        # Should be 1000x300 <= image_dimensions <= 2000x500px
        'DIMENSIONS': {
            'gte': (1000, 300),
            'lte': (2000, 500),
        },

        # width must be >= 30px and less than 60px
        # height must be less than 90px and not equal to 40px
        'DIMENSIONS': {
            'w': {
                'gt': 30,
                'lt': 60,
            },
            'h': {
                'lt': 90,
                'ne': 40,
            }
        },


        # By format (jpeg, png, tiff etc)
        # Uploaded image should be JPEG
        'FORMAT': 'jpeg',

        # Uploaded image should be one of the following
        'FORMAT': ['jpeg', 'png', 'gif'],

        # Uploaded image should not be a GIF
        'FORMAT': {
            'ne': 'gif',
        },

        # Uploaded image should be neither a GIF nor a PNG
        'FORMAT': {
            'ne': ['gif', 'png'],
            'err': 'Wrong image <em>format</em>!'
```

(continues on next page)

(continued from previous page)

```
        },
    }
}
```

1.2 Features

- An image may be validated against its *size (KB)*, *dimensions (px)*, *format (jpeg, png etc)* and *aspect ratio (width/height ratio)*.
- *Well formatted error messages*. They have the form of:

  [IMAGE RULE_NAME] Validation error: **image_value** does not meet validation rule: **rule**.
- Humanized error messages. All rules and image values are *humanized*:
  - `'SIZE': {'gte': 100}` becomes `greater than or equal to 100KB` when rendered
  - `'DIMENSIONS': {'ne': (100, 100)}` becomes `not equal to 100 x 100px` when rendered
- *Overridable error messages*. The default error messages may be overridden by defining an `err` key inside the validation rules:

  `'SIZE': {'gte': 100, 'err': 'Custom error'}` becomes `Custom error` when rendered
- *HTML-safe (custom) error messages*. All error messages (the default or your own) are passed through the function `mark_safe()`.
- Cascading validation rules. It's possible to define a generic rule to some `ImageField` fields of an app and then define another set of rules to a specific `ImageField` field. Common rules will override the generic ones and any new rules will be added to the specific `ImageField` field

```
myapp.models: {
    'SIZE': {
        'lt': 120,
    },
    'FORMAT': 'jpeg',
    'DIMENSIONS': {
        'lt': (500, 600),
    }
 },
 myapp.models.MyModel.img: {
    'DIMENSIONS': (1000, 500),
 },
```

  In the example above (the order does not matter), all `ImageFields` should be `less than 120KB`, `JPEG` images **and** `less than 500 x 600px`. However, the `myapp.models.MyModel.img` field should be `less than 120KB`, `JPEG` image **and** `equal to 1000 x 500px`.

1.3 Running Tests

Does the code actually work?

```
source <YOURVIRTUALENV>/bin/activate
(myenv) $ pip install tox
(myenv) $ tox
```

1.4 Future additions

- Validation of image mode (whether the uploaded image is in indexed mode, greyscale mode etc) based on image's mode. This is quite easy to implement but rather a *rare* validation requirement. Thus, it'll be implemented if users want to validate the mode of the image (which again, it's rare for the web).
- If you think of any other validation (apart from svg) that may be applied to an image and it's not included in this package, please feel free to submit an issue or a PR.

1.5 Credits

Tools used in rendering this package:

- Cookiecutter
- cookiecutter-djangopackage

CHAPTER 2

Installation

You should **always** use a virtualenv or the recommended Python way, pipenv. Whichever works best for you. Once you're inside your preferred virtual environment, run:

```
(venv-name)$ pip install django-vimage
```

CHAPTER 3

Usage

To use django-vimage in a project, add it to your `INSTALLED_APPS`

```
INSTALLED_APPS = (
    ...
    'vimage.apps.VimageConfig',
    ...
)
```

And then define a `VIMAGE` dictionary with the appropriate key:value pairs. Every *key* should be a `str` while every *value* should be a `dict`.

```
VIMAGE = {
    # key:value pairs
}
```

3.1 `VIMAGE` key

Each `VIMAGE` key should be a `str`, the dotted path to one of the following:

- `models` module (i.e `myapp.models`). This is the global setting. The rule will apply to all `ImageField` fields defined in this `models` module.
- Django `Model` (i.e `myapp.models.MyModel`). This is a model-specific setting. The rule will apply to all `ImageField` fields defined under this model.
- Django `ImageField` field (i.e `myapp.models.MyModel.img`). This is a field-specific setting. The rule will apply to just this `ImageField`.

It is allowed to have multiple keys refer to the same app. Keep in mind, though, that keys referring to specific `ImageField`'s have higher precedence to those referring to a specific `Model` and any common rules will be overridden while new ones will be added.

For example, suppose you have a project structure like the following:

```
my_project/
  my_app/
    models.py
    views.py
  my_project/
    settings.py
    urls.py
  manage.py
```

and `my_app.models` defines the following models:

```
from django.db import models

class Planet(models.Model):
    # ... other model fields here
    large_photo = models.ImageField(upload_to='planets')
    small_photo = models.ImageField(upload_to='planets')

class Satellite(models.Model):
    # ... other model fields here
    outer_photo = models.ImageField(upload_to='satellite')
    inner_photo = models.ImageField(upload_to='satellite')
```

and the keys defined are the following:

```
VIMAGE = {
    'my_app.models': {# rules here},
    'my_app.models.Planet': {# rules here},
    'my_app.models.Satellite': {# rules here},
    'my_app.models.Satellite.inner_photo': {# rules here},
}
```

Then, all `ImageField`'s of `my_app` app (`large_photo`, `small_photo`, `outer_photo` and `inner_photo`) will have the rules defined in `my_app.models` dict value. However, the rules defined in `my_app.models.Planet` (affecting `ImageField`'s of the `Planet` model) will override the previous ones and any new will be added. The same principle applies to the `Satellite` `ImageField`'s.

In general, rules have specificity, just like CSS. This is a good thing because you can apply some rules globally and then become more particular on a per `ImageField` level.

The specificity is shown below:

Key	Specificity
`<myapp>.models`	1
`<myapp>.models.<Model>`	2
`<myapp>.models.<Model>.<ImageField>`	3

The higher the specificity, the higher the precedence of the rule.

3.2 `VIMAGE` value

Each `VIMAGE` value should be a dictionary. The structure must be:

```
{
    '<validation_string>': <validation_rule>,
}
```

Each key of the dictionary should be one of the following validation strings:

- *'SIZE'*, image file size
- *'DIMENSIONS'*, image dimensions
- *'FORMAT'*, image format (i.e JPEG, PNG etc)
- *'ASPECT_RATIO'* image width / image height ratio

Depending on the validation string, the corresponding value type (and unit) varies. The table below shows the valid key:value pair types:

Key (always `str`)	Value type	Unit
`'SIZE'`	`<int>|<dict>`	`KB`
`'DIMENSIONS'`	`<tuple>|<list>|<dict>`	`px`
`'FORMAT'`	`<str>|<list>|<dict>`	no unit
`'ASPECT_RATIO'`	`<float>|<dict>`	no unit

For example, the following (full example) rule states that the uploaded image (via the Django Admin) must be, for some reason, equal to 100KB:

```
VIMAGE = {
    'my_app.models.MyModel.img': {
        'SIZE': 100,
    }
}
```

The following rule states that the uploaded image must be either a JPEG or a PNG format:

```
VIMAGE = {
    'my_app.models.MyModel.img': {
        'FORMAT': ['jpeg', 'png'],
    }
}
```

When the value is a dict, `VIMAGE` uses the `operator` module to apply the rules. All keys accept the `<dict>` value type with the following strings as keys:

Listing 1: valid operator strings

```
 +-------------------------+----------------------------+
 | Operator string         | Meaning                    |
 +=========================+============================+
 | 'gte'                   | greater than or equal to   |
 +-------------------------+----------------------------+
 | 'lte'                   | less than or equal to      |
 +-------------------------+----------------------------+
 | 'gt'                    | greater than               |
 +-------------------------+----------------------------+
 | 'lt'                    | less than                  |
 +-------------------------+----------------------------+
 | 'eq'                    | equal to                   |
```

(continues on next page)

(continued from previous page)

```
+------------------------+----------------------------+
| 'ne'                   | not equal to               |
+------------------------+----------------------------+
```

However, the `'FORMAT'` validation rule accepts a *minimal* set of operators that may be applied only to string values (not numbers). That is, `'eq'` and `'ne'`.

Note: Keep in mind that an error is raised if some specific operator pairs are used, for example, `'gte'` and `'eq'`. This is because it makes no sense for an image to be `greater than or equal to something` and at the same time `equal to something`!

Confused? Take a look at the *examples*.

CHAPTER 4

Configuration

Each of the following strings are valid as a dict `key` of the `VIMAGE` dict `value`. Confused? Take a look at a *definition example* of `VIMAGE`.

Note: The developer may provide a custom error which will be automatically HTML escaped. The string may also be *translated*. In order to do that, the value of the validation string must be a `dict` and the key of the custom error should be `'err'`. Example:

```
from django.utils.translation import gettext_lazy as _

VIMAGE = {
    'myapp.models': {
        'SIZE': {
            'lt': 200,
            'err': _('Size should be less than <strong>200KB!</strong>'),
        },
    }
}
```

Note: If `'err'` is not defined then a well-looking default error will appear.

[IMAGE {rule_name}] Validation error: **{value}** does not meet validation rule: **{rule}**.

1. `{rule_name}` is replaced by the corresponding validation string
2. `{value}` is replaced by the corresponding image value under test
3. `{rule}` is replaced by the corresponding rule in a *humanized* form

4.1 'SIZE'

The 'SIZE' key corresponds to the image's file size (measured in KB). It accepts two kind of value types: int or dict.

- If it's an int (must be a positive integer) then it is assumed that the file size of the uploaded image will be **equal** to the value defined.

Listing 1: 'SIZE' with int as value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image file size should be equal to 100KB
        'SIZE': 100,
    }
}
```

- If it's a dict, then any str from *operator strings table* will be valid as long as it's value is an int (positive integer). Also, take a look at this *note*.

Listing 2: 'SIZE' with dict as value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image file size should be less than 200KB
        # and greater than 20KB
        'SIZE': {
            'lt': 200,
            'gt': 20,
            'err': 'custom error here'  # optional
        },
    }
}
```

4.2 'DIMENSIONS'

The 'DIMENSIONS' key corresponds to the image's dimensions, width and height (measured in px). It accepts three kind of value types: tuple, list or dict.

- If it's a tuple (two-length tuple with positive integers) then it is assumed that the dimensions of the uploaded image will be **equal** to the value (tuple) defined ((width, height)).

Listing 3: 'DIMENSIONS' with tuple as value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image dimensions should be equal to 800 x 600px
        # width == 800 and height == 600px
        'DIMENSIONS': (800, 600),
    }
}
```

- If it's a list (one or more two-length tuples with positive integers) then it is assumed that the dimensions of the uploaded image will be **equal** to one of the values defined in the list.

Listing 4: 'DIMENSIONS' with `list` as value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image dimensions should be equal to one of the
        # following: 800x600px, 500x640px or 100x100px.
        'DIMENSIONS': [(800, 600), (500, 640), (100, 100)],
    }
}
```

- If it's a `dict`, then there are two cases. Either use *operator strings table* for keys and a two-length tuple of positive integers for values or use the strings `'w'` and/or `'h'` for keys and (another) `dict` for the value of each one using *operator strings table* for keys and a positive integer for values. Confused? Below are two examples that cover each case.

Listing 5: 'DIMENSIONS' with `dict` as value and tuples as sub-values

```
VIMAGE = {
    'myapp.models': {
        # uploaded image dimensions should be less than 1920x1080px
        # and greater than 800x768px.
        'DIMENSIONS': {
            'lt': (1920, 1080),
            'gt': (800, 768),
            'err': 'custom error here', # optional
        },
    }
}
```

Listing 6: 'DIMENSIONS' with `dict` as value and `'w'`, `'h'` as sub-keys

```
VIMAGE = {
    'myapp.models': {
        # uploaded image width should not be equal to 800px and
        # height should be greater than 600px.
        'DIMENSIONS': {
            'w': {
                'ne': 800,  # set rule just for width
                'err': 'custom error here', # optional
            },
            'h': {
                'gt': 600,  # set rule just for height
                'err': 'custom error here', # optional
            }
        },
    }
}
```

Note: For custom error to work when defining both `'w'` and `'h'`, the `'err'` entry should be placed to both `'w'` and `'h'` dicts.

4.3 'FORMAT'

The 'FORMAT' key corresponds to the image's format (it doesn't have a measure unit since it's just a string), i.e 'jpeg', 'png', 'webp' etc. Taking into account what image formats the browsers support VIMAGE allows the most used formats for the web, which are: 'jpeg', 'png', 'gif', 'bmp' and 'webp'. It accepts three kind of value types: str, list or dict.

- If it's a str then it is assumed that the format of the uploaded image will be **equal** to the value (str) defined.

Listing 7: 'FORMAT' with str as value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image format should be 'jpeg'
        'FORMAT': 'jpeg',
    }
}
```

- If it's a list (list of strings) then it is assumed that the format of the uploaded image will be **equal** to one of the values defined in the list.

Listing 8: 'FORMAT' with list as value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image format should be one of the following:
        # 'jpeg', 'png' or 'webp'.
        'FORMAT': ['jpeg', 'png', 'webp']
    }
}
```

- If it's a dict, then the keys must be either 'eq' or 'ne' (since the other operators cannot apply to str values) and as for the values they may be either a list or a str.

Listing 9: 'FORMAT' with dict as value and str as sub-value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image format should not be 'png'.
        'FORMAT': {
            'ne': 'png',
            'err': 'custom error here', # optional
        },
    }
}
```

Listing 10: 'FORMAT' with dict as value and list as sub-value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image format should not be equal to
        # neither `webp` nor 'bmp'.
        'FORMAT': {
            'ne': ['webp', 'bmp'],
            'err': 'custom error here', # optional
        },
    }
```

(continues on next page)

(continued from previous page)

```
}
```

4.4 'ASPECT_RATIO'

The `'ASPECT_RATIO'` key corresponds to the image's width to height ratio (it doesn't have a measure unit since it's just a decimal number). It accepts two kind of value types: `float` or `dict`.

- If it's a `float` (positive) then it is assumed that the aspect ratio of the uploaded image will be **equal** to the value (`float`) defined.

Listing 11: 'ASPECT_RATIO' with `float` as value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image aspect ratio should be equal to 1.2
        'ASPECT_RATIO': 1.2,
    }
}
```

- If it's a `dict`, then any `str` from *operator strings table* will be valid as long as it's value is a positive `float`. Also, take a look at this *note*.

Listing 12: 'ASPECT_RATIO' with `dict` as value

```
VIMAGE = {
    'myapp.models': {
        # uploaded image aspect ratio should be less than 1.2
        'ASPECT_RATIO': {
            'lt': 2.1,
            'err': 'custom error here', # optional
        },
    }
}
```

If you are a *table-person* maybe this will help you:

Table 1: Summarized table between validation strings and their dict values

Key	Value type
`'SIZE'`	`<int>` - image's file size should be equal to this number `<dict>` - <operator_str>: `<int>`

`'DIMENSIONS'`	`<tuple>` - a two-length tuple of positive integers
	`<list>` - a list of two-length tuples of positive integers
	`<dict>` - <operator_str>: `<tuple>`
	`<dict>` - `'w'` and/or `'h'`: `<dict>` - <operator_str>: `<int>`

`'FORMAT'`	`<str>` - one of `'jpeg'`, `'png'`, `'gif'`, `'bmp'`, `'webp'` `<list>` - a list with one or more of the valid formats `<dict>` - `'ne'` or `'eq'`: `<str>` (one of the valid formats) `<dict>` - `'ne'` or `'eq'`: `<list>` (a list with one or more of the valid formats)
`'ASPECT_RATIO'`	`<float>` - a float number `<dict>` - <operator_str>: `<int>`

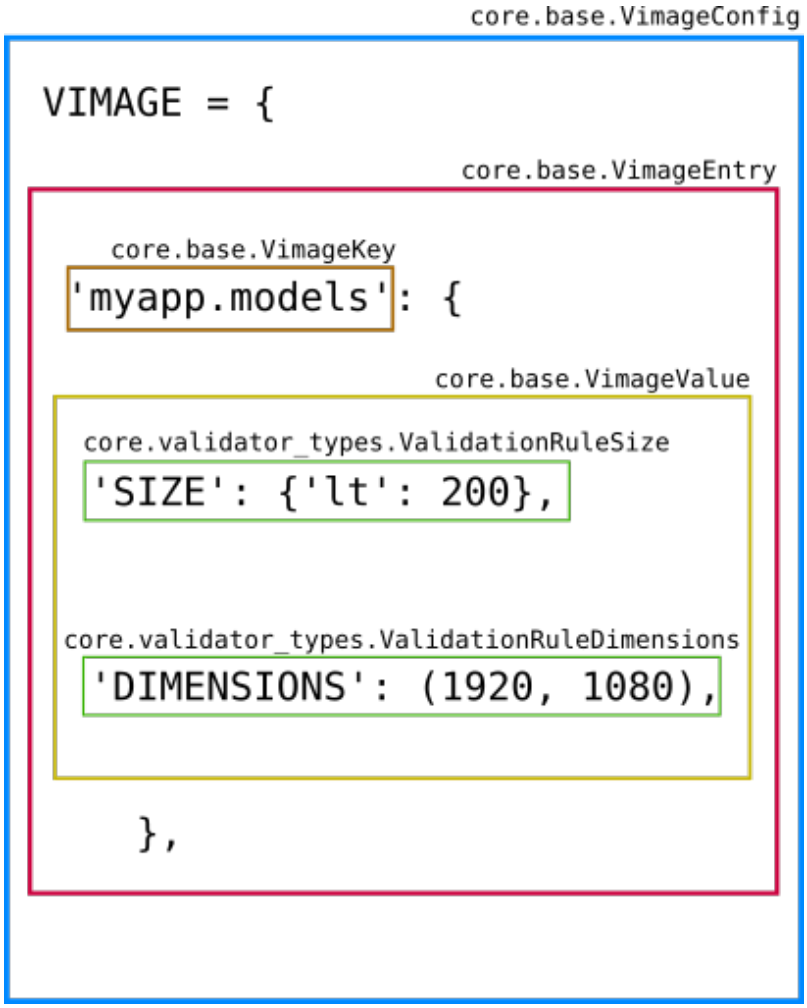
CHAPTER 5

How it works

The mechanism, under the hood, of the `VIMAGE` is pretty simple.

In a glance it does the following:

- *converts* each rule dict (i.e `{'SIZE': 100}` etc) to a corresponding `class`.
- each `class` defines a method which returns a function (callable, the validator)
- a registry is build which has the `ImageField` as keys and a list of functions (validators) as the value
- finally, each validator is added to the `ImageField`'s `validators` attribute (a method defined as a `cached_property`)



For more info about Model `validators` refer to validators.

CHAPTER 6

Examples

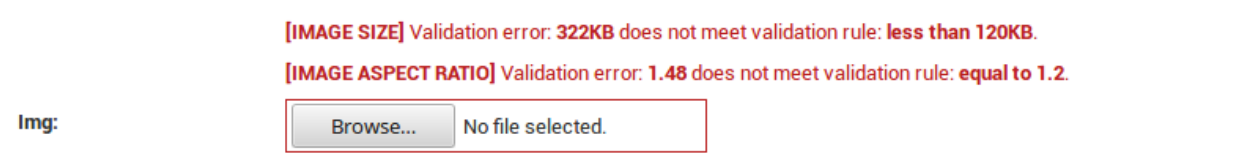
The following examples try to cover a variety of `VIMAGE` *usages* by combining different specificity values and showing their effect on the appropriate fields.

6.1 With specificity 1

The validation rule below will be applied to all `ImageField`'s of the `my_app` app. Each image should be *less than 120KB* **and** *have aspect ratio equal to 1.2*.

```
VIMAGE = {
    'my_app.models': {
        'SIZE': {
            'lt': 120,
        }
        'ASPECT_RATIO': 1.2,
    },
}
```

If we try to upload an image which is more than 120KB and it's aspect ratio is not 1.2 then we'll get the following default error:



Since `'SIZE'` is a `dict` we can define an `'err'` key and give it a *custom error*:

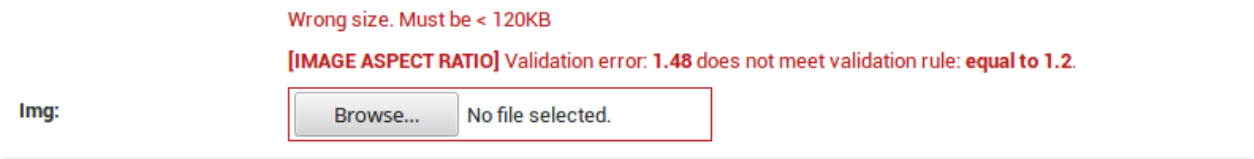
```
VIMAGE = {
    'my_app.models': {
        'SIZE': {
            'lt': 120,
```

(continues on next page)

(continued from previous page)

```
            'err': 'Wrong size. Must be < 120KB',
        }
        'ASPECT_RATIO': 1.2,
    },
}
```

which yields:



6.2 With specificity 2

The validation rule below will be applied to all `ImageField`'s of the `MyModel` model. Each image should be *a JPEG image*, *equal to 400 x 500px* **and** *less than 200KB*.

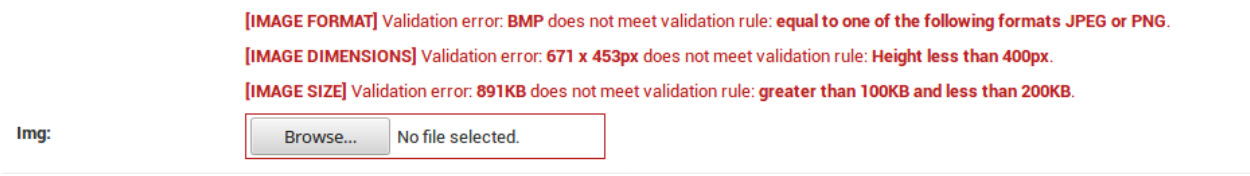
```
VIMAGE = {
    'my_app.models.MyModel': {
        'FORMAT': 'jpeg',
        'DIMENSIONS': (400, 500),
        'SIZE': {
            'lt': 200,
        },
    },
}
```

6.3 With specificity 3

The validation rule below will be applied only to the `img` `ImageField` field. It should *be a JPEG or a PNG image*, *the height should be less than 400px* **and** *be greater than 100KB but less than 200KB*.

```
VIMAGE = {
    'my_app.models.MyModel.img': {
        'FORMAT': ['jpeg', 'png'],
        'DIMENSIONS': {
            'h': {
                'lt': 400,
            }
        },
        'SIZE': {
            'gt': 100,
            'lt': 200,
        },
    },
}
```

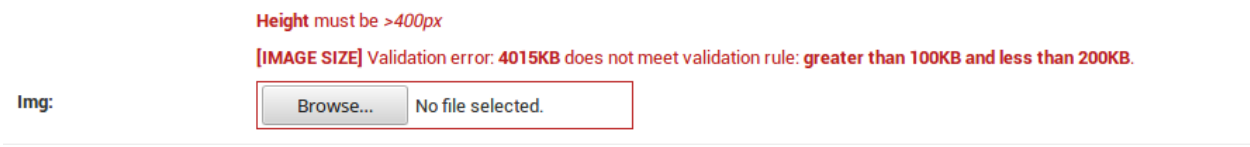
Trying to save the object with an *invalid* image, we get the following default error:



A custom error on `'h'` (height) may be declared, as follows:

```
VIMAGE = {
    'my_app.models.MyModel': {
        'FORMAT': ['jpeg', 'png'],
        'DIMENSIONS': {
            'h': {
                'lt': 400,
                'err': '<strong>Height</strong> must be <em>>400px</em>',
            }
        },
        'SIZE': {
            'gt': 100,
            'lt': 200,
        },
    },
}
```

Trying with an *invalid* image, we get (note that we have provided a valid image format, so the `'FORMAT'` validation passes and not shown):



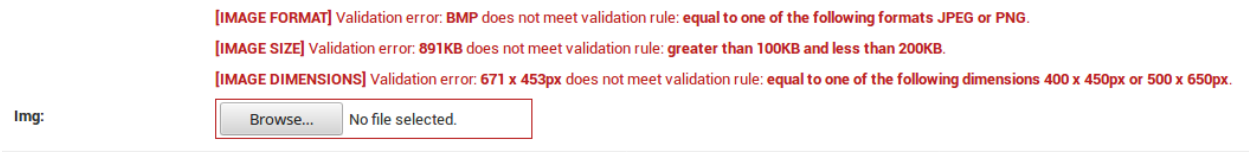
6.4 With specificity 1 + 2

```
VIMAGE = {
    # specificity 1
    'my_app.models': {
        'FORMAT': ['jpeg', 'png'],
        'SIZE': {
            'gt': 100,
            'lt': 200,
        },
    },
    # specificity 2
    'my_app.models.ModelOne': {
        'DIMENSIONS': [(400, 450), (500, 650)],
    },
    # specificity 2
    'my_app.models.ModelTwo': {
        'FORMAT': ['webp'],
    },
}
```

After declaring the above validation rule, the following rules will apply:

all `ImageField`'s of the	Rules
`ModelOne` model	• `'FORMAT': ['jpeg', 'png']` • `'SIZE': {'gt': 100, 'lt': 200}` • `'DIMENSIONS': [(400, 450), (500, 650)]`
`ModelTwo` model	• `'FORMAT': ['webp']` • `'SIZE': {'gt': 100, 'lt': 200}`

and providing (again) an *invalid* image, we get the following default error for the `img` `ImageField` inside the `ModelOne` model:



6.5 With specificity 1 + 3

```
VIMAGE = {
    # specificity 1
    'my_app.models': {
        'DIMENSIONS': {
            'lte': (1920, 1080),
        },
        'FORMAT': 'jpeg',
        'SIZE': {
            'gt': 100,
            'lt': 200,
        },
    },
    # specificity 3
    'my_app.models.ModelOne.img': {
        'DIMENSIONS': (800, 1020),
    },
}
```

After declaring the above validation rule, the following rules will apply:

Fields	Rules
all `ImageField`'s of the `my_app` app	• `'DIMENSIONS': {'lte': (1920, 1080)}` • `'FORMAT': 'jpeg'` • `'SIZE': {'gt': 100, 'lt': 200}`
only the `img` field	• `'DIMENSIONS': (800, 1020)` • `'FORMAT': 'jpeg'` • `'SIZE': {'gt': 100, 'lt': 200}`

CHAPTER 7

Reference

The following sections show some modules of this package, in alphabetical order, just for reference. It's **not** a public API to use. The developer, should only set the *VIMAGE* dictionary.

7.1 vimage.core.base

7.2 vimage.core.checker

7.3 vimage.core.const

7.4 vimage.core.validator_types

CHAPTER 8

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:

8.1 Types of Contributions

8.1.1 Report Bugs

Report bugs at https://github.com/manikos/django-vimage/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.

8.1.2 Fix Bugs

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

8.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

8.1.4 Extend translations

The only languages that the default validation error appears is in English and Greek. You may pull request translations in order to extend the `locale/` dir to other languages too! The number of strings that need to be translated is small, so you won't spend too much time.

8.1.5 Write Documentation

django-vimage could always use more documentation, whether as part of the official django-vimage docs, in docstrings, or even on the web in blog posts, articles, and such.

8.1.6 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/manikos/django-vimage/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 :)

8.2 Get Started!

Ready to contribute? Here's how to set up *django-vimage* for local development.

1. Fork the *django-vimage* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/django-vimage.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv django-vimage
$ cd django-vimage/
$ python setup.py develop
```

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:

```
$ flake8 vimage tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

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.

8.3 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 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/manikos/django-vimage/pull_requests and make sure that the tests pass for all supported Python versions.

8.4 Tips

To run a subset of tests:

```
$ python runtests.py test_checker  # will run only tests/test_suites/test_checker.py
```

CHAPTER 9

Credits

9.1 Development Lead

- Nick Mavrakis <mavrakis.n@gmail.com>

9.2 Contributors

None yet. Why not be the first?

CHAPTER 10

History

10.1 0.1.0 (2018-04-17)

- First release on PyPI

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

V

vimage.core, 25

Index

V

`vimage.core` (*module*), 25