django-versatileimagefield Documentation

Release 1.0

Jonathan Ellenberger

Apr 18, 2017

Contents

A drop-in replacement for django's `ImageField` that provides a flexible, intuitive and *easily-extensible* interface for creating new images from the one assigned to the field.

Click here for a quick overview of what it is, how it works and whether or not it's the right fit for your project.

CHAPTER 1

Compatibility

- Python:
  - 2.7
  - 3.4
  - 3.5
  - 3.6

Note: The 1.2 release dropped support for Python 3.3.x.

- Django:
  - 1.7.x
  - 1.8.x
  - 1.9.x
  - 1.10.x
  - 1.11.x

Note: The 1.4 release dropped support for Django 1.5.x & 1.6.x.

Note: The 1.7 release dropped support for Django 1.7.x.

- Pillow >=2.4.0,<=4.0.0
- Django REST Framework:
  - 2.3.14
  - 2.4.4

- 3.0.x
- 3.1.x
- 3.2.x
- 3.3.x (NOTE: Django REST Framework 3.3.x is not compatible with Django<=1.6.x)
- 3.4.x (NOTE: Django REST Framework 3.4.x is not compatible with Django<=1.6.x)
- 3.5.x (NOTE: Django REST Framework 3.4.x is not compatible with Django<=1.7.x)
- 3.6.x

CHAPTER 2

Code

`django-versatileimagefield` is hosted on github.

CHAPTER 3

Table of Contents

Overview

You're probably using an ImageField.

```
from django.db import models

class ExampleModel(models.Model):
    image = models.ImageField(
        'Image',
        upload_to='images/'
    )
```

You should *swap it out* for a `VersatileImageField`. It's better!

```
from django.db import models

from versatileimagefield.fields import VersatileImageField

class ExampleModel(models.Model):
    image = VersatileImageField(
        'Image',
        upload_to='images/'
    )
```

Works just like `ImageField`

Out-of-the-box, `VersatileImageField` provides the same functionality as `ImageField`:

Template Code	Image
`<img src="{{ instance.image.url }}" />`	

So what sets it apart?

Create Images Wherever You Need Them

A `VersatileImageField` can create new images on-demand **both in templates and the shell**.

Let's make a thumbnail image that would fit within a 200px by 200px area:

Template Code	Image
`<img src="{{ instance.image.thumbnail.200x200 }}" />`	

No crufty templatetags necessary! Here's how you'd do the same in the shell:

```
>>> from someapp.models import ExampleModel
>>> instance = ExampleModel.objects.all()[0]
>>> instance.image.thumbnail['200x200'].url
'/media/__sized__/images/test-image-thumbnail-200x200.jpg'
>>> instance.image.thumbnail['200x200'].name
'__sized__/images/test-image-thumbnail-200x200.jpg'
```

Crop images at specific sizes

You can use it to create cropped images, too:

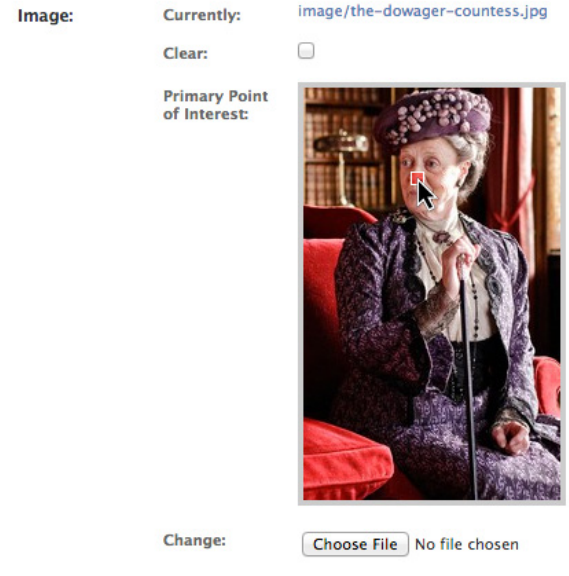
Template Code	Default, Absolutely Centered Crop
`<img src="{{ instance.image.crop.400x400 }}" />`	

Uh-oh. That looks weird.

Custom, Per-Image Cropping

Don't worry! `VersatileImageField` ships with a handy admin-compatible widget that you can use to specify an image's *Primary Point of Interest (PPOI)* by clicking on it.

Note the translucent red square underneath the mouse cursor in the image within the left column below:

Admin Widget PPOI Selection Tool	Resultant Cropped Image
	

Ahhhhh, that's better.

Filters, too!

`VersatileImageField` has *filters*, too! Let's create an inverted image:

Template Code	Image
`<img src="{{ instance.image.filters.invert.url }}" />`	

You can chain filters and sizers together:

Template Code	Image
`<img src="{{ instance.image.filters.invert.thumbnail.200x200 }}" />`	

Write your own Sizers & Filters

Making new sizers and filters (or overriding existing ones) is super-easy via the *Sizer and Filter framework*.

Django REST Framework Integration

If you've got an API powered by Django REST Framework you can use `VersatileImageField` to serve multiple images (in any number of sizes and renditions) from a single field. *Learn more here*.

Flexible in development, light-weight in production

`VersatileImageField`'s on-demand image creation provides maximum flexibility during development but can be *easily turned off* so your app performs like a champ in production.

Fully Tested & Python 3 Ready

`django-versatileimagefield` is a rock solid, fully-tested Django app that is compatible with Python 2.7, 3.4 and 3.5 and works with Django 1.5.x thru 1.9.x

Get Started

You should totally *try it out*! It's 100% backwards compatible with `ImageField` so you've got nothing to lose!

Installation

Installation is easy with pip:

```
$ pip install django-versatileimagefield
```

Python Compatibility

- 2.7.x
- 3.3.x
- 3.4.x
- 3.5.x
- 3.6.x

Django Compatibility

- 1.8.x
- 1.9.x
- 1.10.x
- 1.11.x

Dependencies

- `Pillow` >= 2.4.x

`django-versatileimagefield` depends on the excellent Pillow fork of `PIL`. If you already have PIL installed, it is recommended you uninstall it prior to installing `django-versatileimagefield`:

```
$ pip uninstall PIL
$ pip install django-versatileimagefield
```

Note: `django-versatileimagefield` will not install `django`.

Settings

After installation completes, add `'versatileimagefield'` to `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    # All your other apps here
    'versatileimagefield',
)
```

VERSATILEIMAGEFIELD_SETTINGS

A dictionary that allows you to fine-tune how `django-versatileimagefield` works:

```
VERSATILEIMAGEFIELD_SETTINGS = {
    # The amount of time, in seconds, that references to created images
    # should be stored in the cache. Defaults to `2592000` (30 days)
    'cache_length': 2592000,
    # The name of the cache you'd like `django-versatileimagefield` to use.
    # Defaults to 'versatileimagefield_cache'. If no cache exists with the name
```

```
    # provided, the 'default' cache will be used instead.
    'cache_name': 'versatileimagefield_cache',
    # The save quality of modified JPEG images. More info here:
    # https://pillow.readthedocs.io/en/latest/handbook/image-file-formats.html#jpeg
    # Defaults to 70
    'jpeg_resize_quality': 70,
    # The name of the top-level folder within storage classes to save all
    # sized images. Defaults to '__sized__'
    'sized_directory_name': '__sized__',
    # The name of the directory to save all filtered images within.
    # Defaults to '__filtered__':
    'filtered_directory_name': '__filtered__',
    # The name of the directory to save placeholder images within.
    # Defaults to '__placeholder__':
    'placeholder_directory_name': '__placeholder__',
    # Whether or not to create new images on-the-fly. Set this to `False` for
    # speedy performance but don't forget to 'pre-warm' to ensure they're
    # created and available at the appropriate URL.
    'create_images_on_demand': True,
    # A dot-notated python path string to a function that processes sized
    # image keys. Typically used to md5-ify the 'image key' portion of the
    # filename, giving each a uniform length.
    # `django-versatileimagefield` ships with two post processors:
    # 1. 'versatileimagefield.processors.md5' Returns a full length (32 char)
    #    md5 hash of `image_key`.
    # 2. 'versatileimagefield.processors.md5_16' Returns the first 16 chars
    #    of the 32 character md5 hash of `image_key`.
    # By default, image_keys are unprocessed. To write your own processor,
    # just define a function (that can be imported from your project's
    # python path) that takes a single argument, `image_key` and returns
    # a string.
    'image_key_post_processor': None,
    # Whether to create progressive JPEGs. Read more about progressive JPEGs
    # here: https://optimus.io/support/progressive-jpeg/
    'progressive_jpeg': False
}
```

VERSATILEIMAGEFIELD_USE_PLACEHOLDIT

A boolean that signifies whether optional (`blank=True`) `VersatileImageField` fields that do not *specify a placeholder image* should return placehold.it URLs.

VERSATILEIMAGEFIELD_RENDITION_KEY_SETS

A dictionary used to specify 'Rendition Key Sets' that are used for both *serialization* or as a way to *'warm' image files* so they don't need to be created on demand (i.e. when `settings.VERSATILEIMAGEFIELD_SETTINGS['create_images_on_demand']` is set to `False`) which will greatly improve the overall performance of your app. Here's an example:

```
VERSATILEIMAGEFIELD_RENDITION_KEY_SETS = {
    'image_gallery': [
        ('gallery_large', 'crop__800x450'),
        ('gallery_square_small', 'crop__50x50')
    ],
    'primary_image_detail': [
```

```
        ('hero', 'crop__600x283'),
        ('social', 'thumbnail__800x800')
    ],
    'primary_image_list': [
        ('list', 'crop__400x225'),
    ],
    'headshot': [
        ('headshot_small', 'crop__150x175'),
    ]
}
```

Each key in `VERSATILEIMAGEFIELD_RENDITION_KEY_SETS` signifies a 'Rendition Key Set', a list comprised of 2-tuples wherein the first position is a serialization-friendly name of an image rendition and the second position is a 'Rendition Key' (which dictates how the original image should be modified).

Writing Rendition Keys

Rendition Keys are intuitive and easy to write, simply swap in double-underscores for the dot-notated paths you'd use *in the shell* or *in templates*. Examples:

Intended image	As 'Rendition Key'	In the shell	In templates
400px by 400px Crop	`'crop__400x400'`	`instance.image_field.crop['400x400'].url`	`{{ instance.image_field.crop.400x400 }}`
100px by 100px Thumbnail	`'thumbnail__100x100'`	`instance.image_field.thumbnail['100x100'].url`	`{{ instance.image_field.thumbnail.100x100 }}`
Inverted Image (Full Size)	`'filters__invert'`	`instance.image_field.filters.invert.url`	`{{ instance.image_field.filters.invert }}`
Inverted Image, 50px by 50px crop	`'filters__invert__crop__50x50'`	`instance.image_field.filters.invert.crop['50x50'].url`	`{{ instance.image_field.filters.invert.crop.50x50 }}`

Using Rendition Key Sets

Rendition Key sets are useful! Read up on how they can help you...

- ... *serialize VersatileImageField instances* with Django REST Framework.
- ... *'pre-warm' images to improve performance*.

Model Integration

The centerpiece of `django-versatileimagefield` is its `VersatileImageField` which provides a simple, flexible interface for creating new images from the image you assign to it.

`VersatileImageField` extends django's `ImageField` and can be used as a drop-in replacement for it. Here's a simple example model that depicts a typical usage of django's `ImageField`:

```
# models.py with `ImageField`
from django.db import models

class ImageExampleModel(models.Model):
    name = models.CharField(
        'Name',
        max_length=80
    )
    image = models.ImageField(
        'Image',
        upload_to='images/testimagemodel/',
        width_field='width',
        height_field='height'
    )
    height = models.PositiveIntegerField(
        'Image Height',
        blank=True,
        null=True
    )
    width = models.PositiveIntegerField(
        'Image Width',
        blank=True,
        null=True
    )

    class Meta:
        verbose_name = 'Image Example'
        verbose_name_plural = 'Image Examples'
```

And here's that same model using `VersatileImageField` instead (see highlighted section in the code block below):

```
# models.py with `VersatileImageField`
from django.db import models

from versatileimagefield.fields import VersatileImageField

class ImageExampleModel(models.Model):
    name = models.CharField(
        'Name',
        max_length=80
    )
    image = VersatileImageField(
        'Image',
        upload_to='images/testimagemodel/',
        width_field='width',
        height_field='height'
    )
    height = models.PositiveIntegerField(
        'Image Height',
        blank=True,
        null=True
    )
    width = models.PositiveIntegerField(
        'Image Width',
```

```
        blank=True,
        null=True
    )

    class Meta:
        verbose_name = 'Image Example'
        verbose_name_plural = 'Image Examples'
```

Note: `VersatileImageField` is fully interchangable with django.db.models.ImageField which means you can revert back anytime you'd like. It's fully-compatible with south so migrate to your heart's content!

Specifying Placeholder Images

For `VersatileImageField` fields that are set to `blank=True` you can optionally specify a placeholder image to be used when its sizers and filters are accessed (like a generic silouette for a non-existant user profile image, for instance).

You have two options for specifying placeholder images:

1. `OnDiscPlaceholderImage`: If you want to use an image stored on the same disc as your project's codebase.
2. `OnStoragePlaceholderImage`: If you want to use an image that can be accessed directly with a django storage class.

Note: All placeholder images are transferred-to and served-from the storage class of their associated field.

OnDiscPlaceholderImage

A placeholder image that is stored on the same disc as your project's codebase. Let's add a new, optional `VersatileImageField` to our example model to demonstrate:

```
# models.py
import os

from django.db import models

from versatileimagefield.fields import VersatileImageField
from versatileimagefield.placeholder import OnDiscPlaceholderImage

class ImageExampleModel(models.Model):
    name = models.CharField(
        'Name',
        max_length=80
    )
    image = VersatileImageField(
        'Image',
        upload_to='images/testimagemodel/',
        width_field='width',
        height_field='height'
    )
    height = models.PositiveIntegerField(
```

```
        'Image Height',
        blank=True,
        null=True
    )
    width = models.PositiveIntegerField(
        'Image Width',
        blank=True,
        null=True
    )
    optional_image = VersatileImageField(
        'Optional Image',
        upload_to='images/testimagemodel/optional/',
        blank=True,
        placeholder_image=OnDiscPlaceholderImage(
            path=os.path.join(
                os.path.dirname(os.path.abspath(__file__)),
                'placeholder.gif'
            )
        )
    )

    class Meta:
        verbose_name = 'Image Example'
        verbose_name_plural = 'Image Examples'
```

Note: In the above example the `os` library was used to determine the on-disc path of an image (`placeholder.gif`) that was stored in the same directory as `models.py`.

Where `OnDiscPlaceholderImage` saves images to

All placeholder images are automatically saved into the same storage as the field they are associated with into a top-level-on-storage directory named by the `VERSATILEIMAGEFIELD_SETTINGS['placeholder_directory_name']` setting (defaults to `'__placeholder__'` *docs*).

Placeholder images defined by `OnDiscPlaceholderImage` will simply be saved into the placeholder directory (defaults to `'__placeholder__'` *docs*). The placeholder image defined in the example above would be saved to `'__placeholder__/placeholder.gif'`.

`OnStoragePlaceholderImage`

A placeholder image that can be accessed with a django storage class. Example:

```
# models.py
from django.db import models

from versatileimagefield.fields import VersatileImageField
from versatileimagefield.placeholder import OnStoragePlaceholderImage

class ImageExampleModel(models.Model):
    name = models.CharField(
        'Name',
        max_length=80
```

```
    )
    image = VersatileImageField(
        'Image',
        upload_to='images/testimagemodel/',
        width_field='width',
        height_field='height'
    )
    height = models.PositiveIntegerField(
        'Image Height',
        blank=True,
        null=True
    )
    width = models.PositiveIntegerField(
        'Image Width',
        blank=True,
        null=True
    )
    optional_image = VersatileImageField(
        'Optional Image',
        upload_to='images/testimagemodel/optional/',
        blank=True,
        placeholder_image=OnStoragePlaceholderImage(
            path='images/placeholder.gif'
        )
    )

    class Meta:
        verbose_name = 'Image Example'
        verbose_name_plural = 'Image Examples'
```

By default, `OnStoragePlaceholderImage` will look look for this image in your default storage class (as determined by default_storage) but you can explicitly specify a custom storage class with the optional keyword argument `storage`:

```
# models.py
from django.db import models

from versatileimagefield.fields import VersatileImageField
from versatileimagefield.placeholder import OnStoragePlaceholderImage

from .storage import CustomStorageCls

class ImageExampleModel(models.Model):
    name = models.CharField(
        'Name',
        max_length=80
    )
    image = VersatileImageField(
        'Image',
        upload_to='images/testimagemodel/',
        width_field='width',
        height_field='height'
    )
    height = models.PositiveIntegerField(
        'Image Height',
        blank=True,
        null=True
    )
```

```
    width = models.PositiveIntegerField(
        'Image Width',
        blank=True,
        null=True
    )
    optional_image = VersatileImageField(
        'Optional Image',
        upload_to='images/testimagemodel/optional/',
        blank=True,
        placeholder_image=OnStoragePlaceholderImage(
            path='images/placeholder.gif',
            storage=CustomStorageCls()
        )
    )

    class Meta:
        verbose_name = 'Image Example'
        verbose_name_plural = 'Image Examples'
```

Where `OnStoragePlaceholderImage` saves images to

Placeholder images defined by `OnStoragePlaceholderImage` will be saved into the placeholder directory (defaults to `'__placeholder__'` *docs*) within the same folder heirarchy as their original storage class. The placeholder image used in the example above would be saved to `'__placeholder__/image/placeholder.gif`.

Specifying a Primary Point of Interest (PPOI)

The *crop Sizer* is super-useful for creating images at a specific size/aspect-ratio however, sometimes you want the 'crop centerpoint' to be somewhere other than the center of a particular image. In fact, the initial inspiration for `django-versatileimagefield` came as a result of tackling this very problem.

The `crop` Sizer's core functionality (located in the `versatileimagefield.versatileimagefield.CroppedImage.crop_on_centerpoint` method) was inspired by PIL's ImageOps.fit function (by Kevin Cazabon) which takes an optional keyword argument, `centering`, that expects a 2-tuple comprised of floats which are greater than or equal to 0 and less than or equal to 1. These two values together form a cartesian coordinate system which dictates the percentage of pixels to 'trim' off each of the long sides (i.e. left/right or top/bottom, depending on the aspect ratio of the cropped size vs. the original size):

	Left	Center	Right
Top	`(0.0, 0.0)`	`(0.0, 0.5)`	`(0.0, 1.0)`
Middle	`(0.5, 0.0)`	`(0.5, 0.5)`	`(0.5, 1.0)`
Bottom	`(1.0, 0.0)`	`(1.0, 0.5)`	`(1.0, 1.0)`

The `crop` Sizer works in a similar way but converts the 2-tuple into an exact (x, y) pixel coordinate which is then used as the 'centerpoint' of the crop. This approach gives significantly more accurate results than using `ImageOps.fit`, especially when dealing with PPOI values located near the edges of an image *or* aspect ratios that differ significantly from the original image.

Note: Even though the PPOI value is used as a crop 'centerpoint', the pixel it corresponds to won't necessarily be in the center of the cropped image, especially if its near the edges of the original image.

Note: At present, only the `crop` Sizer changes how it creates images based on PPOI but a `VersatileImageField` makes its PPOI value available to ALL its attached Filters and Sizers. Get creative!

The PPOIField

Each image managed by a `VersatileImageField` can store its own, unique PPOI in the database via the easy-to-use `PPOIField`. Here's how to integrate it into our example model (relevant lines highlighted in the code block below):

```
# models.py with `VersatileImageField` & `PPOIField`
from django.db import models

from versatileimagefield.fields import VersatileImageField, \
    PPOIField

class ImageExampleModel(models.Model):
    name = models.CharField(
        'Name',
        max_length=80
    )
    image = VersatileImageField(
        'Image',
        upload_to='images/testimagemodel/',
        width_field='width',
        height_field='height',
        ppoi_field='ppoi'
    )
    height = models.PositiveIntegerField(
        'Image Height',
        blank=True,
        null=True
    )
    width = models.PositiveIntegerField(
        'Image Width',
        blank=True,
        null=True
    )
    ppoi = PPOIField(
        'Image PPOI'
    )

    class Meta:
        verbose_name = 'Image Example'
        verbose_name_plural = 'Image Examples'
```

As you can see, you'll need to add a new `PPOIField` field to your model and then include the name of that field in the `VersatileImageField`'s `ppoi_field` keyword argument. That's it!

Note: `PPOIField` is fully-compatible with south so migrate to your heart's content!

How PPOI is Stored in the Database

The **Primary Point of Interest** is stored in the database as a string with the x and y coordinates limited to two decimal places and separated by an 'x' (for instance: `'0.5x0.5'` or `'0.62x0.28'`).

Setting PPOI

PPOI is set via the `ppoi` attribute on a `VersatileImageField`..

When you save a model instance, `VersatileImageField` will ensure its currently-assigned PPOI value is 'sent' to the `PPOIField` associated with it (if any) prior to writing to the database.

Via The Shell

```
# Importing our example Model
>>> from someapp.models import ImageExampleModel
# Retrieving a model instance
>>> example = ImageExampleModel.objects.all()[0]
# Retrieving the current PPOI value associated with the image field
# A `VersatileImageField`'s PPOI value is ALWAYS associated with the `ppoi`
# attribute, irregardless of what you named the `PPOIField` attribute on your model
>>> example.image.ppoi
(0.5, 0.5)
# Creating a cropped image
>>> example.image.crop['400x400'].url
u'/media/__sized__/images/testimagemodel/test-image-crop-c0-5__0-5-400x400.jpg'
# Changing the PPOI value
>>> example.image.ppoi = (1, 1)
# Creating a new cropped image with the new PPOI value
>>> example.image.crop['400x400'].url
u'/media/__sized__/images/testimagemodel/test-image-crop-c1__1-400x400.jpg'
# PPOI values can be set as either a tuple or a string
>>> example.image.ppoi = '0.1x0.55'
>>> example.image.ppoi
(0.1, 0.55)
>>> example.image.ppoi = (0.75, 0.25)
>>> example.image.crop['400x400'].url
u'/media/__sized__/images/testimagemodel/test-image-crop-c0-75__0-25-400x400.jpg'
# u'0.75x0.25' is written to the database in the 'ppoi' column associated with
# our example model
>>> example.save()
```

As you can see, changing an image's PPOI changes the filename of the cropped image. This ensures updates to a `VersatileImageField`'s PPOI value will result in unique cache entries for each unique image it creates.

Note: Each time a field's PPOI is set, its attached Filters & Sizers will be immediately updated with the new value.

FormField/Admin Integration

It's pretty hard to accurately set a particular image's PPOI when working in the Python shell so `django-versatileimagefield` ships with an admin-ready formfield. Simply add an image, click 'Save and continue editing', click where you'd like the PPOI to be and then save your model instance again. A helpful translucent red square will indicate where the PPOI value is currently set to on the image:

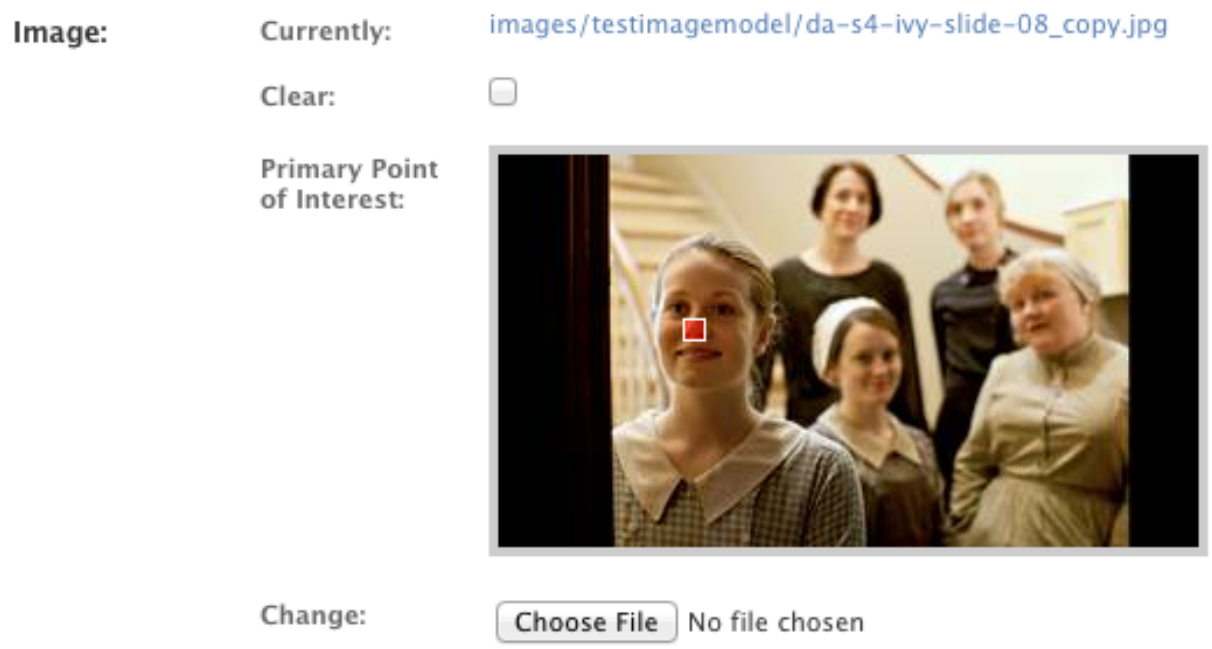


Fig. 3.1: django-versatileimagefield PPOI admin widget example

Note: `PPOIField` is not editable so it will be automatically excluded from the admin.

Django 1.5 Admin Integration for required `VersatileImageField` fields

If you're using a required (i.e. `blank=False`) `VersatileImageField` on a project running Django 1.5 you'll need a custom form class to circumvent an already-fixed-in-Django-1.6 issue (that has to do with required fields associated with a `MultiValueField`/`MultiWidget` used in a `ModelForm`).

The example below uses an example model `YourModel` that has a required `VersatileImageField` as the `image` attribute.

```
# yourapp/forms.py

from django.forms import ModelForm

from versatileimagefield.fields import SizedImageCenterpointClickDjangoAdminField

from .models import YourModel

class YourModelForm(VersatileImageTestModelForm):
    image = SizedImageCenterpointClickDjangoAdminField(required=False)

    class Meta:
        model = YourModel
        fields = ('image',)
```

Note the `required=False` in the formfield definition in the above example.

Integrating the custom form into the admin:

```
# yourapp/admin.py

from django.contrib import admin

from .forms import YourModelForm
from .models import YourModel

class YourModelAdmin(admin.ModelAdmin):
    form = YourModelForm

admin.site.register(YourModel, YourModelAdmin)
```

Using Sizers and Filters

Where `VersatileImageField` shines is in its ability to create new images on the fly via its Sizer & Filter framework.

Sizers

Sizers provide a way to create new images of differing sizes from the one assigned to the field. `VersatileImageField` ships with two Sizers, `thumbnail` and `crop`.

Each Sizer registered to the *Sizer registry* is available as an attribute on each `VersatileImageField`. Sizers are `dict` subclasses that only accept precisely formatted keys comprised of two integers – representing width and height, respectively – separated by an 'x' (i.e. `['400x400']`). If you send a malformed/invalid key to a Sizer, a `MalformedSizedImageKey` exception will raise.

Included Sizers

thumbnail

Here's how you would create a thumbnail image that would be constrained to fit within a 400px by 400px area:

```
# Importing our example Model
>>> from someapp.models import ImageExampleModel
# Retrieving a model instance
>>> example = ImageExampleModel.objects.all()[0]
# Displaying the path-on-storage of the image currently assigned to the field
>>> example.image.name
u'images/testimagemodel/test-image.jpg'
# Retrieving the path on the field's storage class to a 400px wide
# by 400px tall constrained thumbnail of the image.
>>> example.image.thumbnail['400x400'].name
u'__sized__/images/testimagemodel/test-image-thumbnail-400x400.jpg'
# Retrieving the URL to the 400px wide by 400px tall thumbnail
>>> example.image.thumbnail['400x400'].url
u'/media/__sized__/images/testimagemodel/test-image-thumbnail-400x400.jpg'
```

Note: Images are created on-demand. If no image had yet existed at the location required – by either the path (`.name`) *or* URL (`.url`) shown in the highlighted lines above – one would have been created directly before returning

it.

Here's how you'd open the thumbnail image we just created as an image file directly in the shell:

```
>>> thumbnail_image = example.image.field.storage.open(
...     example.image.thumbnail['400x400'].name
... )
```

crop

To create images cropped to a specific size, use the `crop` Sizer:

```
# Retrieving the URL to a 400px wide by 400px tall crop of the image
>>> example.image.crop['400x400'].url
u'/media/__sized__/images/testimagemodel/test-image-crop-c0-5__0-5-400x400.jpg'
```

The `crop` Sizer will first scale an image down to its longest side and then crop/trim inwards, centered on the **Primary Point of Interest** (PPOI, for short). For more info about what PPOI is and how it's used see the *Specifying a Primary Point of Interest (PPOI)* section.

How Sized Image Files are Named/Stored

All Sizers subclass from `versatileimagefield.datastructures.sizedimage.SizedImage` which uses a unique-to-size-specified string – provided via its `get_filename_key()` method – that is included in the filename of each image it creates.

Note: The `thumbnail` Sizer simply combines `'thumbnail'` with the size key passed (i.e. `'400x400'`) while the `crop` Sizer combines `'crop'`, the field's PPOI value (as a string) and the size key passed; all Sizer 'filename keys' begin and end with dashes `'-'` for readability.

All images created by a Sizer are stored within the field's `storage` class in a top-level folder named `'__sized__'`, maintaining the same descendant folder structure as the original image. If you'd like to change the name of this folder to something other than `'__sized__'`, adjust the value of `VERSATILEIMAGEFIELD_SETTINGS['sized_directory_name']` within your settings file.

Sizers are quick and easy to write, for more information about how it's done, see the *Writing a Custom Sizer* section.

Filters

Filters create new images that are the same size and aspect ratio as the original image.

Included Filters

invert

The `invert` filter will invert the color palette of an image:

```
# Importing our example Model
>>> from someapp.models import ImageExampleModel
# Retrieving a model instance
>>> example = ImageExampleModel.objects.all()[0]
# Returning the path-on-storage to the image currently assigned to the field
>>> example.image.name
u'images/testimagemodel/test-image.jpg'
# Displaying the path (within the field's storage class) to an image
# with an inverted color pallete from that of the original image
>>> example.image.filters.invert.name
u'images/testimagemodel/__filtered__/test-image__invert__.jpg'
# Displaying the URL to the inverted image
>>> example.image.filters.invert.url
u'/media/images/testimagemodel/__filtered__/test-image__invert__.jpg'
```

As you can see, there's a `filters` attribute available on each `VersatileImageField` which contains all filters currently registered to the Filter registry.

Using Sizers with Filters

What makes Filters extra-useful is that they have access to all registered Sizers:

```
# Creating a thumbnail of a filtered image
>>> example.image.filters.invert.thumbnail['400x400'].url
u'/media/__sized__/images/testimagemodel/__filtered__/test-image__invert__-thumbnail-
↪400x400.jpg'
# Creating a crop from a filtered image
>>> example.image.filters.invert.crop['400x400'].url
u'/media/__sized__/images/testimagemodel/__filtered__/test-image__invert__-c0-5__0-5-
↪400x400.jpg'
```

Note: Filtered images are created the first time they are directly accessed (by either evaluating their `name`/`url` attributes or by accessing a Sizer attached to it). Once created, a reference is stored in the cache for each created image which makes for speedy subsequent retrievals.

How Filtered Image Files are Named/Stored

All Filters subclass from `versatileimagefield.datastructures.filteredimage.FilteredImage` which provides a `get_filename_key()` method that returns a unique-to-filter-specified string – surrounded by double underscores, i.e. `'__invert__'` – which is appended to the filename of each image it creates.

All images created by a Filter are stored within a folder named `__filtered__` that sits in the same directory as the original image. If you'd like to change the name of this folder to something other than '**filtered**', adjust the value of `VERSATILEIMAGEFIELD_SETTINGS['filtered_directory_name']` within your settings file.

Filters are quick and easy to write, for more information about creating your own, see the *Writing a Custom Filter* section.

Using Sizers / Filters in Templates

Template usage is straight forward and easy since both attributes and dictionary keys can be accessed via dot-notation; no crufty templatetags necessary:

```
<!-- Sizers -->
<img src="{{ instance.image.thumbnail.400x400 }}" />
<img src="{{ instance.image.crop.400x400 }}" />

<!-- Filters -->
<img src="{{ instance.image.filters.invert.url }}" />

<!-- Filters + Sizers -->
<img src="{{ instance.image.filters.invert.thumbnail.400x400 }}" />
<img src="{{ instance.image.filters.invert.crop.400x400 }}" />
```

Note: Using the `url` attribute on Sizers is optional in templates. Why? All Sizers return an instance of `versatileimagefield.datastructures.sizedimage.SizedImageInstance` which provides the sized image's URL via the `__unicode__()` method (which django's templating engine looks for when asked to render class instances directly).

Writing Custom Sizers and Filters

It's quick and easy to create new Sizers and Filters for use on your project's `VersatileImageField` fields or *modify already-registered Sizers and Filters*.

Both Sizers and Filters subclass from `versatileimagefield.datastructures.base.ProcessedImage` which provides a *preprocessing API* as well as all the business logic necessary to retrieve and save images.

The 'meat' of each Sizer & Filter – what actually modifies the original image – resides within the `process_image` method which all subclasses must define (not doing so will raise a `NotImplementedError`). Sizers and Filters expect slightly different keyword arguments (Sizers required `width` and `height`, for example) see below for specifics:

Writing a Custom Sizer

All Sizers should subclass `versatileimagefield.datastructures.sizedimage.SizedImage` and, at a minimum, MUST do two things:

1. Define either the `filename_key` attribute or override the `get_filename_key()` method which is necessary for creating unique-to-Sizer-and-size-specified filenames. If neither of the aforementioned is done a `NotImplementedError` exception will be raised.
2. Define a `process_image` method that accepts the following arguments:
    - `image`: a PIL Image instance
    - `image_format`: A valid image mime type (e.g. 'image/jpeg'). This is provided by the `create_resized_image` method (which calls `process_image`).
    - `save_kwargs`: A `dict` of any keyword arguments needed by PIL's `Image.save` method (initially provided by the pre-processing API).
    - `width`: An integer representing the width specified by the user in the size key.

- `height`: An integer representing the height specified by the user in the size key.

For an example, let's take a look at the `thumbnail` Sizer (`versatileimagefield.versatileimagefield.ThumbnailImage`):

```
from django.utils.six import BytesIO

from PIL import Image

from .datastructures import SizedImage

class ThumbnailImage(SizedImage):
    """
    Sizes an image down to fit within a bounding box

    See the `process_image()` method for more information
    """

    filename_key = 'thumbnail'

    def process_image(self, image, image_format, save_kwargs,
                      width, height):
        """
        Returns a BytesIO instance of `image` that will fit
        within a bounding box as specified by `width`x`height`
        """
        imagefile = BytesIO()
        image.thumbnail(
            (width, height),
            Image.ANTIALIAS
        )
        image.save(
            imagefile,
            **save_kwargs
        )
        return imagefile
```

Important: `process_image` should *always* return a `BytesIO` instance. See *What process_image should return* for more information.

Ensuring Sized Images Can Be Deleted

If your `SizedImage` subclass uses more than just `filename_key` to construct filenames than you'll also want to define the `filename_key_regex` attribute.

Confused? Let's take a look at `CroppedImage` – which includes individual image PPOI values in the images it creates – as an example:

```
class CroppedImage(SizedImage):
    """
    A SizedImage subclass that creates a 'cropped' image.

    See the `process_image` method for more details.
    """

    filename_key = 'crop'
```

```
    filename_key_regex = r'crop-c[0-9-]+__[0-9-]+'

    def get_filename_key(self):
        """Return the filename key for cropped images."""
        return "{key}-c{ppoi}".format(
            key=self.filename_key,
            ppoi=self.ppoi_as_str()
        )
```

The `get_filename_key` method above is what is used by the sizer to create a filename fragment when **creating** images. It combines the `filename_key` with an individual image's PPOI value which ensures PPOI changes result in newly created images (which makes sense when you're cropping in respect to PPOI). The `filename_key_regex` is a regular expression pattern utilized by the *file deletion API* in order to find cropped images created from the original image.

Writing a Custom Filter

All Filters should subclass `versatileimagefield.datastructures.filteredimage.FilteredImage` and only need to define a `process_image` method with following arguments:

- `image`: a PIL Image instance
- `image_format`: A valid image mime type (e.g. 'image/jpeg'). This is provided by the `create_resized_image()` method (which calls `process_image`).
- `save_kwargs`: A `dict` of any keyword arguments needed by PIL's `Image.save` method (initially provided by the pre-processing API).

For an example, let's take a look at the `invert` Filter (`versatileimagefield.versatileimagefield.InvertImage`):

```
from django.utils.six import BytesIO

from PIL import ImageOps

from .datastructures import FilteredImage

class InvertImage(FilteredImage):
    """
    Inverts the colors of an image.

    See the `process_image()` for more specifics
    """

    def process_image(self, image, image_format, save_kwargs={}):
        """
        Returns a BytesIO instance of `image` with inverted colors
        """
        imagefile = BytesIO()
        inv_image = ImageOps.invert(image)
        inv_image.save(
            imagefile,
            **save_kwargs
        )
        return imagefile
```

Important: `process_image` should **always** return a `BytesIO` instance. See *What process_image should return* for more information.

What `process_image` should return

Any `process_image` method you write should *always* return a `BytesIO` instance comprised of raw image data. The actual image file will be written to your field's storage class via the `save_image` method. Note how `save_kwargs` is passed into PIL's `Image.save` method in the examples above, this ensures PIL knows how to write this data (based on mime type or any other per-filetype specific options provided by the *preprocessing API*).

The Pre-processing API

Both Sizers and Filters have access to a pre-processing API that provides hooks for doing any per-mime-type processing. This allows your Sizers and Filters to do one thing for JPEGs and another for GIFs, for instance. One example of this is in how Sizers 'know' how to preserve transparency for GIFs or save JPEGs as RGB (at the user-defined quality):

```
# versatileimagefield/datastructures/sizedimage.py
class SizedImage(ProcessedImage, dict):
    "<a bunch of ommited code here>"

    def preprocess_GIF(self, image, **kwargs):
        """
        Receives a PIL Image instance of a GIF and returns 2-tuple:
            * [0]: Original Image instance (passed to `image`)
            * [1]: Dict with a transparency key (to GIF transparency layer)
        """
        return (image, {'transparency': image.info['transparency']})

    def preprocess_JPEG(self, image, **kwargs):
        """
        Receives a PIL Image instance of a JPEG and returns 2-tuple:
            * [0]: Image instance, converted to RGB
            * [1]: Dict with a quality key (mapped to the value of `QUAL` as
                   defined by the `VERSATILEIMAGEFIELD_JPEG_RESIZE_QUALITY`
                   setting)
        """
        if image.mode != 'RGB':
            image = image.convert('RGB')
        return (image, {'quality': QUAL})
```

All pre-processors should accept one required argument `image` (A PIL Image instance) and `**kwargs` (for easy extension by subclasses) and return a 2-tuple of the image and a dict of any additional keyword arguments to pass along to PIL's `Image.save` method.

Pre-processor Naming Convention

In order for preprocessor methods to run, they need to be named correctly via this simple naming convention: `preprocess_FILETYPE`. Here's a list of all currently-supported file types:

- BMP
- DCX

- EPS
- GIF
- JPEG
- PCD
- PCX
- PDF
- PNG
- PPM
- PSD
- TIFF
- XBM
- XPM

So, if you'd want to write a PNG-specific preprocessor, your Sizer or Filter would need to define a method named `preprocess_PNG`.

Note: I've only tested `VersatileImageField` with PNG, GIF and JPEG files; the list above is what PIL supports, for more information about per filetype support in PIL visit here.

Registering Sizers and Filters

Registering Sizers and Filters is easy and straight-forward; if you've ever registered a model with django's `admin` you'll feel right at home.

`django-versatileimagefield` finds Sizers & Filters within modules named `versatileimagefield` – (i.e. `versatileimagefield.py`) that are available at the 'top level' of each app on `INSTALLED_APPS`.

Here's an example:

```
somedjangoapp/
    __init__.py
    models.py               # Models
    admin.py                # Admin config
    versatilimagefield.py   # Custom Sizers and Filters here
```

After defining your Sizers and Filters you'll need to register them with the `versatileimagefield_registry`. Here's how the `ThumbnailSizer` is registered (see the highlighted lines in the following code block for the relevant bits):

```
# versatileimagefield/versatileimagefield.py
from django.utils.six import BytesIO

from PIL import Image

from .datastructures import SizedImage
from .registry import versatileimagefield_registry


class ThumbnailImage(SizedImage):
```

```
    """
    Sizes an image down to fit within a bounding box

    See the `process_image()` method for more information
    """

    filename_key = 'thumbnail'

    def process_image(self, image, image_format, save_kwargs,
                      width, height):
        """
        Returns a BytesIO instance of `image` that will fit
        within a bounding box as specified by `width`x`height`
        """
        imagefile = BytesIO()
        image.thumbnail(
            (width, height),
            Image.ANTIALIAS
        )
        image.save(
            imagefile,
            **save_kwargs
        )
        return imagefile

# Registering the ThumbnailSizer to be available on VersatileImageField
# via the `thumbnail` attribute
versatileimagefield_registry.register_sizer('thumbnail', ThumbnailImage)
```

All Sizers are registered via the `versatileimagefield_registry.register_sizer` method. The first argument is the attribute you want to make the Sizer available at and the second is the `SizedImage` subclass.

Filters are just as easy. Here's how the `InvertImage` filter is registered (see the highlighted lines in the following code block for the relevant bits):

```
from django.utils.six import BytesIO

from PIL import ImageOps

from .datastructures import FilteredImage
from .registry import versatileimagefield_registry


class InvertImage(FilteredImage):
    """
    Inverts the colors of an image.

    See the `process_image()` for more specifics
    """

    def process_image(self, image, image_format, save_kwargs={}):
        """
        Returns a BytesIO instance of `image` with inverted colors
        """
        imagefile = BytesIO()
        inv_image = ImageOps.invert(image)
        inv_image.save(
            imagefile,
```

```
            **save_kwargs
        )
        return imagefile

versatileimagefield_registry.register_filter('invert', InvertImage)
```

All Filters are registered via the `versatileimagefield_registry.register_filter` method. The first argument is the attribute you want to make the Filter available at and the second is the FilteredImage subclass.

Unallowed Sizer & Filter Names

Sizer and Filter names cannot begin with an underscore as it would prevent them from being accessible within the template layer. Additionally, since Sizers are available for use directly on a `VersatileImageField`, there are some Sizer names that are unallowed; trying to register a Sizer with one of the following names will result in a `UnallowedSizerName` exception:

- `build_filters_and_sizers`
- `chunks`
- `close`
- `closed`
- `create_on_demand`
- `delete`
- `encoding`
- `field`
- `file`
- `fileno`
- `filters`
- `flush`
- `height`
- `instance`
- `isatty`
- `multiple_chunks`
- `name`
- `newlines`
- `open`
- `path`
- `ppoi`
- `read`
- `readinto`
- `readline`
- `readlines`

- save
- seek
- size
- softspace
- storage
- tell
- truncate
- url
- validate_ppoi
- width
- write
- writelines
- xreadlines

Overriding an existing Sizer or Filter

If you try to register a Sizer or Filter with an attribute name that's already in use (like `crop` or `thumbnail` or `invert`), an `AlreadyRegistered` exception will raise.

> **Caution:** A Sizer can have the same name as a Filter (since names are only required to be unique per type) however it's **not** recommended.

If you'd like to override an already-registered Sizer or Filter just use either the `unregister_sizer` or `unregister_filter` methods of `versatileimagefield_registry`. Here's how you could 'override' the `crop` Sizer:

```
from versatileimagefield.registry import versatileimagefield_registry

# Unregistering the 'crop' Sizer
versatileimagefield_registry.unregister_sizer('crop')
# Registering a custom 'crop' Sizer
versatileimagefield_registry.register_sizer('crop', SomeCustomSizedImageCls)
```

The order that Sizers and Filters register corresponds to their containing app's position on `INSTALLED_APPS`. This means that if you want to override one of the default Sizers or Filters your app needs to be included after `'versatileimagefield'`:

```
# settings.py
INSTALLED_APPS = (
    'versatileimagefield',
    'yourcustomapp'  # This app can override the default Sizers and Filters
)
```

Deleting Created Images

Note: The deletion API was added in version 1.3

`VersatileImageField` ships with a number of useful methods that make it easy to delete unwanted/stale images and/or clear out their associated cache entries.

The docs below all reference this example model:

```
# someapp/models.py

from django.db import models

from versatileimagefield.fields import VersatileImageField

class ExampleImageModel(models.Model):
    image = VersatileImageField(upload_to='images/')
```

Deleting Individual Renditions

Clearing The Cache

To delete a cache entry associated with a created image just call its `clear_cache()` method:

```
>>> from someapp.models import ExampleImageModel
>>> instance = ExampleImageModel.objects.get()
# Deletes the cache entry associated with the 400px by 400px
# crop of instance.image
>>> instance.image.crop['400x400'].clear_cache()
# Deletes the cache entry associated with the inverted
# filter of instance.image
>>> instance.image.filters.invert.clear_cache()
# Deletes the cache entry associated with the inverted + cropped-to
# 400px by 400px rendition of instance.image
>>> instance.image.filters.invert.crop['400x400'].clear_cache()
```

Deleting An Image

To delete a created image just call its `delete()` method:

```
>>> from someapp.models import ExampleImageModel
>>> instance = ExampleImageModel.objects.get()
# Deletes the image AND cache entry associated with the 400px by 400px
# crop of instance.image
>>> instance.image.crop['400x400'].delete()
# Deletes the image AND cache entry associated with the inverted
# filter of instance.image
>>> instance.image.filters.invert.delete()
# Deletes the image AND cache entry associated with the inverted +
# cropped-to 400px by 400px rendition of instance.image
>>> instance.image.filters.invert.crop['400x400'].delete()
```

Note: Deleting an image will also clear its associated cache entry.

Deleting Multiple Renditions

Deleting All Sized Images

To delete all sized images created by a field use its `delete_sized_images` method:

```
>>> from someapp.models import ExampleImageModel
>>> instance = ExampleImageModel.objects.get()
# Deletes all sized images and cache entries associated with instance.image
>>> instance.image.delete_sized_images()
```

Deleting All Filtered Images

To delete all filtered images created by a field use its `delete_filtered_images` method:

```
>>> from someapp.models import ExampleImageModel
>>> instance = ExampleImageModel.objects.get()
# Deletes all filtered images and cache entries associated with instance.image
>>> instance.image.delete_filtered_images()
```

Deleting All Filtered + Sized Images

To delete all filtered + sized images created by a field use its `delete_filtered_sized_images` method:

```
>>> from someapp.models import ExampleImageModel
>>> instance = ExampleImageModel.objects.get()
# Deletes all filtered + sized images and cache entries associated with instance.image
>>> instance.image.delete_filtered_sized_images()
```

Deleting ALL Created Images

To delete ALL images created by a field (sized, filtered & filtered + sized) use its `delete_all_created_images` method:

```
>>> from someapp.models import ExampleImageModel
>>> instance = ExampleImageModel.objects.get()
# Deletes ALL images and cache entries associated with instance.image
>>> instance.image.delete_all_created_images()
```

Note: The original image (`instance.name` on `instance.field.storage` in the above example) will NOT be deleted.

Automating Deletion on `post_delete`

The rendition deleting and cache clearing functionality was written to address the need to delete 'stale' images (i.e. images created from a `VersatileImageField` field on a model instance that has since been deleted). Here's a simple example of how to accomplish that with a `post_delete` signal receiver:

```
# someapp/models.py

from django.db import models
from django.dispatch import receiver

from versatileimagefield.fields import VersatileImageField

class ExampleImageModel(models.Model):
    image = VersatileImageField(upload_to='images/')

@receiver(models.signals.post_delete, sender=ExampleImageModel)
def delete_ExampleImageModel_images(sender, instance, **kwargs):
    """
    Deletes ExampleImageModel image renditions on post_delete.
    """
    # Deletes Image Renditions
    instance.image.delete_all_created_images()
    # Deletes Original Image
    instance.image.delete(save=False)
```

Warning: There's no undo for deleting images off a storage object so proceed at your own risk!

Django REST Framework Integration

If you've got an API powered by Tom Christie's excellent Django REST Framework and want to serve images in multiple sizes/renditions `django-versatileimagefield` has you covered with it's `VersatileImageFieldSerializer`.

Example

To demonstrate how it works we'll use this simple model:

```
# myproject/person/models.py

from django.db import models

from versatileimagefield.fields import VersatileImageField, PPOIField


class Person(models.Model):
    """Represents a person."""
    name_first = models.CharField('First Name', max_length=80)
    name_last = models.CharField('Last Name', max_length=100)
    headshot = VersatileImageField(
        'Headshot',
        upload_to='headshots/',
        ppoi_field='headshot_ppoi'
    )
    headshot_ppoi = PPOIField()

    class Meta:
```

```
        verbose_name = 'Person'
        verbose_name_plural = 'People'
```

OK, let's write a simple `ModelSerializer` subclass to serialize Person instances:

```
# myproject/person/serializers.py

from rest_framework import serializers

from versatileimagefield.serializers import VersatileImageFieldSerializer

from .models import Person


class PersonSerializer(serializers.ModelSerializer):
    """Serializes Person instances"""
    headshot = VersatileImageFieldSerializer(
        sizes=[
            ('full_size', 'url'),
            ('thumbnail', 'thumbnail__100x100'),
            ('medium_square_crop', 'crop__400x400'),
            ('small_square_crop', 'crop__50x50')
        ]
    )

    class Meta:
        model = Person
        fields = (
            'name_first',
            'name_last',
            'headshot'
        )
```

And here's what it would look like serialized:

```
>>> from myproject.person.models import Person
>>> john_doe = Person.objects.create(
...     name_first='John',
...     name_last='Doe',
...     headshot='headshots/john_doe_headshot.jpg'
... )
>>> john_doe.save()
>>> from myproject.person.serializers import PersonSerializer
>>> john_doe_serialized = PersonSerializer(john_doe)
>>> john_doe_serialized.data
{
    'name_first': 'John',
    'name_last': 'Doe',
    'headshot': {
        'full_size': 'http://api.yoursite.com/media/headshots/john_doe_headshot.jpg',
        'thumbnail': 'http://api.yoursite.com/media/headshots/john_doe_headshot-
↪thumbnail-400x400.jpg',
        'medium_square_crop': 'http://api.yoursite.com/media/headshots/john_doe_
↪headshot-crop-c0-5__0-5-400x400.jpg',
        'small_square_crop': 'http://api.yoursite.com/media/headshots/john_doe_
↪headshot-crop-c0-5__0-5-50x50.jpg',
    }
}
```

As you can see, the `sizes` argument on `VersatileImageFieldSerializer` simply unpacks the list of 2-tuples using the value in the first position as the attribute of the image and the second position as a 'Rendition Key' which dictates how the original image should be modified.

Reusing Rendition Key Sets

It's common to want to re-use similar sets of images across models and fields so `django-versatileimagefield` provides a setting, `VERSATILEIMAGEFIELD_RENDITION_KEY_SETS` for defining them (*docs*).

Let's move the Rendition Key Set we used above into our settings file:

```
# myproject/settings.py

VERSATILEIMAGEFIELD_RENDITION_KEY_SETS = {
    'person_headshot': [
        ('full_size', 'url'),
        ('thumbnail', 'thumbnail__100x100'),
        ('medium_square_crop', 'crop__400x400'),
        ('small_square_crop', 'crop__50x50')
    ]
}
```

Now, let's update our serializer to use it:

```
# myproject/person/serializers.py

from rest_framework import serializers

from versatileimagefield.serializers import VersatileImageFieldSerializer

from .models import Person


class PersonSerializer(serializers.ModelSerializer):
    """Serializes Person instances"""
    headshot = VersatileImageFieldSerializer(
        sizes='person_headshot'
    )

    class Meta:
        model = Person
        fields = (
            'name_first',
            'name_last',
            'headshot'
        )
```

That's it! Now that you know how to define Rendition Key Sets, leverage them to *improve performance*!

Improving Performance

During development, `VersatileImageField`'s *on-demand image creation* enables you to quickly iterate but, once your application is deployed into production, this convenience adds a small bit of overhead that you'll probably want to turn off.

Turning off on-demand image creation

To turn off on-demand image creation just set the `'create_images_on_demand'` key of the `VERSATILEIMAGEFIELD_SETTINGS` setting to `False` (*docs*). Now your `VersatileImageField` fields will return URLs to images without first checking to see if they've actually been created yet.

Note: Once an image has been created by a `VersatileImageField`, a reference to it is stored in the cache which makes for speedy subsequent retrievals. Setting `VERSATILEIMAGEFIELD_SETTINGS['create_images_on_demand']` to `False` bypasses this entirely making `VersatileImageField` perform even faster (*docs*).

Ensuring images are created

This boost in performance is great but now you'll need to ensure that the images your application links-to actually exist. Luckily, `VersatileImageFieldWarmer` will help you do just that. Here's an example in the Python shell using the *example model* from the Django REST Framework serialization example:

```
>>> from myproject.person.models import Person
>>> from versatileimagefield.image_warmer import VersatileImageFieldWarmer
>>> person_img_warmer = VersatileImageFieldWarmer(
...     instance_or_queryset=Person.objects.all(),
...     rendition_key_set='person_headshot',
...     image_attr='headshot',
...     verbose=True
... )
>>> num_created, failed_to_create = person_img_warmer.warm()
```

`num_created` will be an integer of how many images were successfully created and `failed_to_create` will be a list of paths to images (on the field's storage class) that could not be created (due to a PIL/Pillow error, for example).

This technique is useful if you've recently converted your project's `models.ImageField` fields to use `VersatileImageField` or if you want to 'pre warm' images as part of a Fabric script.

Note: The above example would create a set of images (as dictated by the `'person_headshot'` *Rendition Key Set*) for the `headshot` field of each `Person` instance. `rendition_key_set` also accepts a valid *Rendition Key Set* directly:

```
>>> person_img_warmer = VersatileImageFieldWarmer(
...     instance_or_queryset=Person.objects.all(),
...     rendition_key_set=[
...         ('large_horiz_crop', '1200x600'),
...         ('large_vert_crop', '600x1200'),
...     ],
...     image_attr='headshot',
...     verbose=True
... )
```

Note: Setting `verbose=True` when instantiating a `VersatileImageFieldWarmer` will display a yum-style progress bar showing the image warming progress:

```
>>> num_created, failed_to_create = person_img_warmer.warm()
[###########----------------------------------------] 20/100 (20%)
```

Note: The `image_attr` argument can be dot-notated in order to follow `ForeignKey` and `OneToOneField` relationships. Example: `'related_model.headshot'`.

Auto-creating sets of images on `post_save`

You also might want to create new images immediately after model instances are saved. Here's how we'd do it with our example model (see highlighted lines below):

```
# myproject/person/models.py

from django.db import models
from django.dispatch import receiver

from versatileimagefield.fields import VersatileImageField, PPOIField
from versatileimagefield.image_warmer import VersatileImageFieldWarmer


class Person(models.Model):
    """Represents a person."""
    name_first = models.CharField('First Name', max_length=80)
    name_last = models.CharField('Last Name', max_length=100)
    headshot = VersatileImageField(
        'Headshot',
        upload_to='headshots/',
        ppoi_field='headshot_ppoi'
    )
    headshot_ppoi = PPOIField()

    class Meta:
        verbose_name = 'Person'
        verbose_name_plural = 'People'

@receiver(models.signals.post_save, sender=Person)
def warm_Person_headshot_images(sender, instance, **kwargs):
    """Ensures Person head shots are created post-save"""
    person_img_warmer = VersatileImageFieldWarmer(
        instance_or_queryset=instance,
        rendition_key_set='person_headshot',
        image_attr='headshot'
    )
    num_created, failed_to_create = person_img_warmer.warm()
```

CHAPTER 4

Release Notes

1.7.0

- Added support for Django 1.11 (thanks, @slurms and @matthiask!).

1.6.3

- Added support for Pillow 4.0 and Python 3.6 (thanks, @aleksihakli!)
- Improved docs for writing custom sizers and filters (thanks, @njamaleddine!)

1.6.2

- Added support for Pillow 3.4.2

1.6.1

- Logs are now created when thumbnail generation fails (thanks, @artursmet!!!).
- Added support for Django 1.10.x and djangorestframework 3.5.x.
- Fixed a bug that caused *delete_all_created_images()* to fail on field instances that didn't have filtered, sized & filtered+sized images.

1.6

- Fixed a bug that prevented sized images from deleting properly when the field they were associated with was using a custom `upload_to` function. If you are using a custom `SizedImage` subclass on your project then be sure to check out *this section* in the docs. (Thanks, @Mortal!)

1.5

- Fixed a bug that was causing placeholder images to serialize incorrectly with `VersatileImageFieldSerializer` (thanks, @romanosipenko!).
- Ensured embedded ICC profiles are preserved when creating new images (thanks, @gbts!).
- Added support for progressive JPEGs (more info *here*).

1.4

- Included JPEG resize quality to sized image keys.
- Added `VERSATILEIMAGEFIELD_SETTINGS['image_key_post_processor']` setting for specifying a function that will post-process sized image keys to create simpler/cleaner filenames. `django-versatileimagefield` ships with two built-in post processors: `'versatileimagefield.processors.md5'` and `'versatileimagefield.processors.md5_16'` (more info *here*).

1.3

- Added the ability to delete images & cache entries created by a `VersatileImageField` both *individually* and *in bulk*. Relevant docs *here*.

1.2.2

- Fixed a critical bug that broke initial project setup (i.e. when `django.setup()` is run) when an app config path was included in `INSTALLED_APPS` (as opposed to a 'vanilla' python module).

1.2.1

- Fixed a bug that caused `collectstatic` to fail when using placeholder images with external storage, like Amazon S3 (thanks, @jelko!).
- `VersatileImageField` now returns its placeholder URL if `.url` is accessed directly (previously only placeholder images were returned if a sizer or filter was accessed). Thanks (again), @jelko!

1.2

- Fixed a bug that caused `collectstatic` to fail when using `ManifestStaticFilesStorage` (thanks, @theskumar!).
- Dropped support for Python 3.3.x.
- Added support for Django 1.9.x.

1.1

- Re-added support for Django 1.5.x (by request, support for Django 1.5.x was previously dropped in the 0.4 release). If you're using `django-versatileimagefield` on a Django 1.5.x project please be sure to read *this bit of documentation*.
- Added support for Django REST Framework 3.3.x.

1.0.6

- Updated `VersatileImageFieldSerializer` to serve image URLs as absolute URIs (if its associated field's storage class isn't doing so already).
  - Formerly: /media/headshots/jane_doe_headshot.jpg
  - Now: http://localhost:8000/media/headshots/jane_doe_headshot.jpg

1.0.5

- Fixed image preview on form validation errors (thanks, @securedirective!).

1.0.4

- Finessed/improved widget functionality for both optional and 'PPOI-less' fields (thanks, @SebCorbin!).

1.0.3

- Addressed Django 1.9 deprecation warnings (`get_cache` and `importlib`)
- Enabled `VersatileImageField` formfield to be overriden via `**kwargs`

1.0.2

- Removed clear checkbox from widgets on required fields.

1.0.1

- Squashed a critical bug in *OnDiscPlaceholderImage*

1.0

- Added support for Django 1.8.
- Numerous documentation edits/improvements.

0.6.2

- Squashed a bug that caused the *javascript 'click' widget* to fail to initialize correctly when multiple VersatileImageFields were displayed on the same page in the admin.
- Added django.contrib.staticfiles integration to widgets.

0.6.1

- Squashed a bug that was throwing an `AttributeError` when uploading new images.

0.6

- Squashed a bug that raised a `ValueError` in the admin when editing a model instance with a `VersatileImageField` that specified `ppoi_field`, `width_field` and `height_field`.
- Admin 'click' widget now works in Firefox.
- `django-versatileimagefield` is now available for installation via wheel.

0.5.4

- Squashed a bug that was causing the admin 'click' widget to intermittently fail
- Simplified requirements installation (which makes django-versatileimagefield installable by `pip<=1.6`)

0.5.3

- Changed `PPOIField` to be `editable=False` by default to address a bug that consistently raised `ValidationError` in ModelForms and the admin

0.5.2

- Squashed a bug that prevented `PPOIField` from serializing correctly

0.5.1

- Squashed an installation bug with `pip` 6+

0.5

- Added Python 3.3 & 3.4 compatibility
- Improved cropping with extreme PPOI values

0.4

- Dropped support for Django 1.5.x
- Introducing per-field *placeholder image* image support! (Note: global placeholder support has been deprecated.)
- Added the `VERSATILEIMAGEFIELD_USE_PLACEHOLDIT` setting (*docs*)

0.3.1

- Squashed a pip installation bug.

0.3

- Added a test suite with Travis CI and coveralls integration.
- Introduced support for *Django REST Framework 3.0* serialization.

0.2.1

- Ensuring *admin widget*-dependent thumbnail images are created even if `VERSATILEIMAGEFIELD_SETTINGS['create_on_demand']` is set to `False`

0.2

- Introduced *Django REST Framework support*!
- Added ability to turn off on-demand image creation and pre-warm images to *improve performance*.

0.1.5

- Squashed `CroppedImage` bug that was causing black stripes to appear on crops of images with PPOI values that were to the right and/or bottom of center (greater-than 0.5).

0.1.4

- Overhauled how `CroppedImage` processes PPOI value when creating cropped images. This new approach yields significantly more accurate results than using the previously utilized `ImageOps.fit` function, especially when dealing with PPOI values located near the edges of an image *or* aspect ratios that differ significantly from the original image.
- Improved PPOI validation
- Squashed unset `VERSATILEIMAGEFIELD_SETTINGS['global_placeholder_image']` bug.
- Set `crop` Sizer default resample to PIL.Image.ANTIALIAS

0.1.3

- Added support for auto-rotation during pre-processing as dictated by 'Orientation' EXIF data, if available.
- Added release notes to docs

0.1.2

- Removed redundant javascript from ppoi 'click' widget (thanks, @theskumar!)

0.1.1

- Converted giant README into Sphinx-friendly RST
- Docs added to readthedocs

0.1

- Initial open source release