django-versatileimagefield Documentation

Release 0.1.1

Jonathan Ellenberger

December 15, 2014

Contents

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

CHAPTER 1

Release Notes

1.1 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).

1.2 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

1.3 0.1.3

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

1.4 0.1.2

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

1.5 0.1.1

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

1.6 0.1

- Initial open source release

In A Nutshell

- Creates images anywhere you need them: not just *in templates*.
- Non-destructive: Your original image is never modified.
- *Sizer and Filter framework*: enables you to quickly add new – or modify existing – ways to create new images:
  - **Sizers** create images with new sizes and/or aspect ratios
  - **Filters** change the appearance of an image
- *Sizers can be chained onto Filters*: Use case: give me a black-and-white, 400px by 400px square crop of this image.
- *Primary Point of Interest (PPOI) support*: provides a way to specify where the 'primary point of interest' of each individual image is – a value which is available to all Sizers and Filters. Use case: sometimes you want the 'crop centerpoint' to be somewhere other than the center of an image. Includes *a user-friendly formfield/widget for selecting PPOI* in the admin (or anywhere else you use ModelForms).
- Works with any storage: Stores the images it creates within the same storage class as your field (just like django's `FileField` & `ImageField`). Works great with a local filesystem or external storage (like Amazon S3).
- Fully interchangeable with `ImageField`: you can easily remove VersatileImageField from your project's models whenever you'd like.
- Integrated caching: References to created images are stored in the cache, keeping your application running quickly and efficiently.

CHAPTER 3

Table of Contents

3.1 Installation

Installation is easy with pip:

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

3.1.1 Dependencies

- `django`>=1.5.0
- `Pillow` >= 2.4.0

`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`.

3.1.2 Settings

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

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

You can fine-tune how `django-versatileimagefield` works via the `VERSATILEIMAGEFIELD_SETTINGS` setting:

```
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:
    # http://pillow.readthedocs.org/en/latest/handbook/image-file-formats.html#jpeg
    # Defaults to 70
    'jpeg_resize_quality': 70,
    # A path on disc to an image that will be used as a 'placeholder'
    # for non-existant images.
    # If 'global_placeholder_image' is unset, the excellent, free-to-use
    # http://placehold.it service will be used instead.
    'global_placeholder_image': '/path/to/an-image.png',
    # 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__'
}
```

3.2 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 to using django's `ImageField` anytime you'd like. It's fully-compatible with south so migrate to your heart's content!

3.3 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` 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 that dictates what 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!

3.3.1 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):

```python
# 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'`).

3.3.2 Setting PPOI

PPOI is set via the `ppoi` attribute on a `VersatileImageField`. You should **always** set an image's PPOI here (as opposed to directly on a `PPOIField` attribute) since a `VersatileImageField` will ensure updated values are passed-down to all its attached Filters & Sizers.

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.

3.3.3 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:

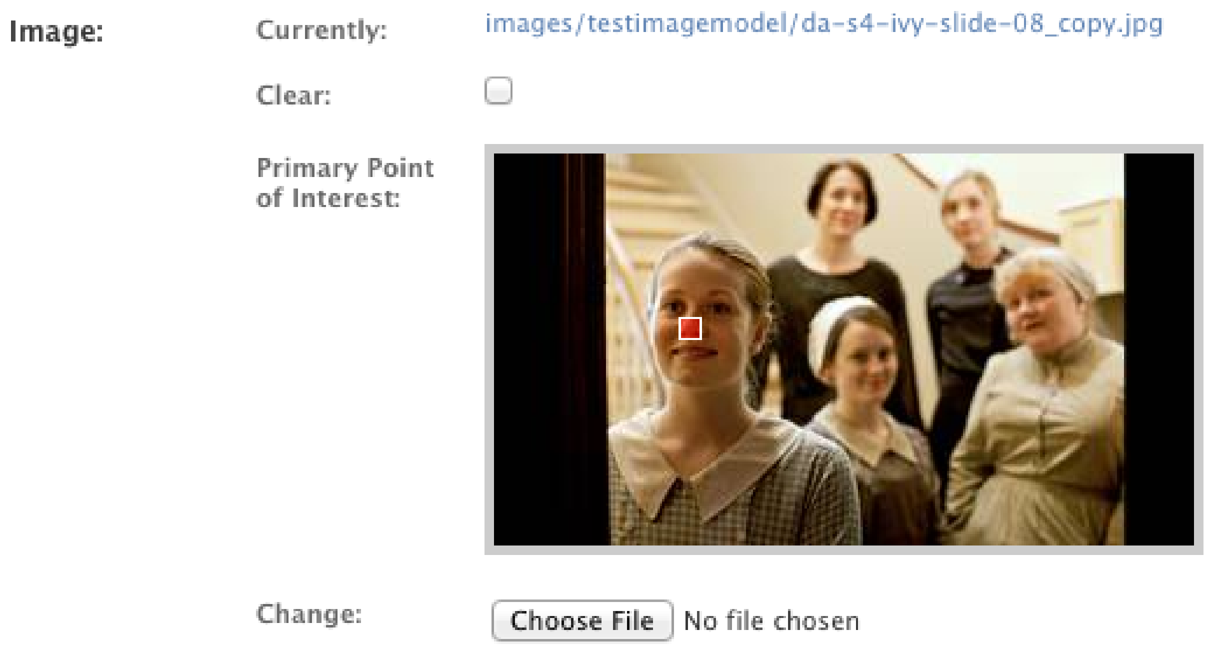
Image:
Currently:
images/testimagemodel/da-s4-ivy-slide-08_copy.jpg
Clear:
Primary Point of Interest:
Change:
Choose File
No file chosen

Figure 3.1: django-versatileimagefield PPOI admin widget example

3.4 Using Sizers and Filters

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

3.4.1 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 them.

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.sizedimag` 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.

3.4.2 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.

3.4.3 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).

3.5 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 – a.k.a what actually modifies the original image – takes place 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:

3.5.1 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.ThumbnailIma`

```
import StringIO

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 StringIO instance of `image` that will fit
        within a bounding box as specified by `width`x`height`
        """
        imagefile = StringIO.StringIO()
        image.thumbnail(
            (width, height),
            Image.ANTIALIAS
        )
        image.save(
            imagefile,
            **save_kwargs
        )
        return imagefile
```

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

3.5.2 Writing a Custom Filter

All Filters should subclass `versatileimagefield.datastructures.filteredimage.FilteredImage` and only need to define a `process_filter` 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`):

```
import StringIO

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 StringIO instance of `image` with inverted colors
        """
        imagefile = StringIO.StringIO()
        inv_image = ImageOps.invert(image)
        inv_image.save(
            imagefile,
            **save_kwargs
        )
        return imagefile
```

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

3.5.3 What `process_image` should return

Any `process_image` method you write should *always* return a `StringIO` 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*).

3.5.4 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.

3.5.5 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
import StringIO

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 StringIO instance of `image` that will fit
        within a bounding box as specified by `width`x`height`
        """
        imagefile = StringIO.StringIO()
        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):

```python
import StringIO

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 StringIO instance of `image` with inverted colors
        """
        imagefile = StringIO.StringIO()
        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`
- `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`

3.5.6 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
)
```

CHAPTER 4

TODO for v0.2

- Tests!
- Placeholder docs
- Programmatically delete images created by `VersatileImageField` (including clearing their connected cache keys)
- Management command for auto-generating sets of images (and pre-warming the cache)
- Templatetags for sizing/filtering static images