django-simpleimages Documentation

Release 1.3.4

Saul Shanabrook

Nov 06, 2018

Contents

This opinionated Django utility will take the image file from one `ImageField` and transform it onto another field, when the model saves.

CHAPTER 1

Why. . . ?

I believe that keeping the implementation of this package as simple as possible. When I say simple, I mean in comparison to other image transformation packages in Django.

1.1 The Alternative

The most popular apps, such as sorl-thumbnail, generate the transformed images in the request-response cycle, in the template or the views. That was the images are never out of date and are not stored in the database, which makes sense because there isn't really anything new that should be stored by a scaled down image. And it makes sense that it is present in the template, because it really is a presentation detail. And it is the easiest method, and can be implemented with a few lines of code. For example:

```
{% thumbnail item.image "100x100" crop="center" as im %}
    <img src="{{ im.url }}" width="{{ im.width }}" height="{{ im.height }}">
{% endthumbnail %}
```

1.1.1 Performance Problems

I ran into performance problems with this approach. Since images are generated in the request-response cycle, caching strategies are essential to minimize database and storage access. I then found django-imagekit, which is much more advanced and allows a great flexibility on every bit of the image generation process. However I still found myself struggling to understand the exact implementation details of how and when the images were generated. This isn't something I should have been worrying about, apart from the fact that some of my pages were timing out generating hundreds of thumbnails.

1.2 My Solution

So I decided to write an implementation that anyone could understand. `django-simpleimages` uses the standard Even though is more verbose, and requires an extra database column, storing transformed images in their own fields presents several advantages. It allows *caching of image dimensions*, using Django's built in solution. It also is easy to understand when the storage backend is being accessed, because you are simply accessing a normal `ImageField`.

CHAPTER 2

Configuration

Add `simpleimages` to your `INSTALLED_APPS` to use the management command.

If you want to transform the images using workers, set the `SIMPLEIMAGES_TRANSFORM_CALLER` to a function that will call the transform function. It defaults to `'simpleimages.callers.default'`, which transforms images synchronously. See `callers` for all provided image transform callers.

The *async docs* section has more details on managing image retrieval for async creation.

2.1 Requirements

- Django 1.5, 1.6, 1.7, 1.8
- Python 2.7, 3.2, 3.3, 3.4, 3.5

CHAPTER 3

Usage

3.1 Models

Here is an example model that will create transformed images on save:

```
from django.db import models
import simpleimages.transforms
import simpleimages.trackers

class YourModel(models.Model):
    image = models.ImageField(
        upload_to='images/'
    )
    thumbnail_image = models.ImageField(
        blank=True,
        null=True,
        editable=False,
        upload_to='transformed_images/thumbnails/'
    )
    large_image = models.ImageField(
        blank=True,
        null=True,
        editable=False,
        upload_to='transformed_images/large/'
    )

    transformed_fields = {
        'image': {
            'thumbnail_image': simpleimages.transforms.Scale(width=10),
            'large_image': simpleimages.transforms.Scale(width=200),
        }
    }

simpleimages.tracking.track_model(YourModel)
```

`track_model()` is called with the model you want to track. When that model is saved, *`perform_transformation()`* uses the `transformed_fields` attribute of the model to determine a mapping of source to destination and transform functions.

See *`transforms`* for all the provided transformations.

3.1.1 Dimension Caching

I would recommend using `height_field` and `width_field` to save the image dimensions. Otherwise (at least with `storages.backends.s3boto.S3BotoStorage`), the file will have to be retrieved once to get `url` and another time to get the dimensions:

```python
import os

from django.db import models

import simpleimages.transforms
import simpleimages.trackers


def image_path_function(subfolder, instance, filename):
    return os.path.join(
        instance.content_name,
        'photos',
        subfolder,
        filename
    )


def original_image_path_function(instance, filename):
    return image_path_function('original', instance, filename)


def thumbnail_image_path_function(instance, filename):
    return image_path_function('thumbnail', instance, filename)


def large_image_path_function(instance, filename):
    return image_path_function('large', instance, filename)


class Photo(models.Model):
    image = models.ImageField(
        upload_to=original_image_path_function,
        max_length=1000,

    )
    thumbnail_image = models.ImageField(
        blank=True,
        null=True,
        editable=False,
        upload_to=thumbnail_image_path_function,
        height_field='thumbnail_image_height',
        width_field='thumbnail_image_width',
        max_length=1000
    )
    large_image = models.ImageField(
```

(continues on next page)

(continued from previous page)

```
        blank=True,
        null=True,
        editable=False,
        upload_to=large_image_path_function,
        height_field='large_image_height',
        width_field='large_image_width',
        max_length=1000
    )
    # cached dimension fields
    thumbnail_image_height = models.PositiveIntegerField(
        null=True,
        blank=True,
        editable=False,
    )
    thumbnail_image_width = models.PositiveIntegerField(
        null=True,
        blank=True,
        editable=False,
    )
    large_image_height = models.PositiveIntegerField(
        null=True,
        blank=True,
        editable=False,
    )
    large_image_width = models.PositiveIntegerField(
        null=True,
        blank=True,
        editable=False,
    )

    transformed_fields = {
        'image': {
            'thumbnail_image': simpleimages.transforms.Scale(height=600),
            'large_image': simpleimages.transforms.Scale(height=800),
        }
    }

simpleimages.trackers.track_model(Photo)
```

3.1.2 Performing Transforms Asynchronously

By default all transformations are performed when the model is saved. If you want to instead perform the transformations asynchronously, for the obvious performance reasons, you by setting `SIMPLEIMAGES_TRANSFORM_CALLER`. Set this to the dotted path to any function that will take the transform function as its first argument and the arguments to call it with as subsequent arguments and keyword arguments. This format was based around django-rq. To perform all transforms through django-rq set `SIMPLEIMAGES_TRANSFORM_CALLER='django_rq.enqueue'`.

There is also built in support for celery, just set `SIMPLEIMAGES_TRANSFORM_CALLER='simpleimages.callers.celery'`

Then you have to account for the fact that sometimes the transformed images won't be available in time to render them on the page. If you want to fall back to the source image, if the transformed image isn't rendered yet, use something like this:

```
import os

from django.db import models

import simpleimages.transforms
import simpleimages.trackers


def image_path_function(subfolder):
    return lambda instance, filename: os.path.join(
        instance.content_name,
        'photos',
        subfolder,
        filename
    )


def original_image_path_function(instance, filename):
    image_path_function('original')(instance, filename)


def thumbnail_image_path_function(instance, filename):
    image_path_function('thumbnail')(instance, filename)


def large_image_path_function(instance, filename):
    image_path_function('large')(instance, filename)


class Photo(models.Model):
    image = models.ImageField(
        upload_to=original_image_path_function,
        max_length=1000,

    )
    thumbnail_image = models.ImageField(
        blank=True,
        null=True,
        editable=False,
        upload_to=thumbnail_image_path_function,
        height_field='thumbnail_image_height',
        width_field='thumbnail_image_width',
        max_length=1000
    )
    large_image = models.ImageField(
        blank=True,
        null=True,
        editable=False,
        upload_to=large_image_path_function,
        height_field='large_image_height',
        width_field='large_image_width',
        max_length=1000
    )
    # cached dimension fields
    thumbnail_image_height = models.PositiveIntegerField(
        null=True,
        blank=True,
```

(continues on next page)

(continued from previous page)

```
        editable=False,
    )
    thumbnail_image_width = models.PositiveIntegerField(
        null=True,
        blank=True,
        editable=False,
    )
    large_image_height = models.PositiveIntegerField(
        null=True,
        blank=True,
        editable=False,
    )
    large_image_width = models.PositiveIntegerField(
        null=True,
        blank=True,
        editable=False,
    )

    @property
    def safe_thumbnail_image(self):
        return self.thumbnail_image or self.image

    @property
    def safe_large_image(self):
        return self.large_image or self.image


    transformed_fields = {
        'image': {
            'thumbnail_image': simpleimages.transforms.Scale(height=600),
            'large_image': simpleimages.transforms.Scale(height=800),
        }
    }

simpleimages.trackers.track_model(Photo)
```

Then access the transformed images with `instance.safe_thumbnail_image` instead.

3.2 Management Command

Since the images are only transformed on the save of the model, if you change the transform function, the instances will not be updated until you resave them. If you want to retransform all the images in a model or app use *management.commands.retransform*

CHAPTER 4

Contributing

If you find issues or would like to see a feature suppored, head over to the issues section: and report it.

To contribute code in any form, fork the github repository: and clone it locally. Create a new branch for your feature:

```
git commit -b feature/whatever-you-like
```

Add proper docstrings to any changed or added code.

Then make sure all the tests past (and write new ones for any new features).

To run the tests:

```
docker-compose up -d db redis
docker-compose run tests
```

Compile the documentation and check if it looks right:

```
docker-compose run tests make docs-html
open docs/build/index.html
```

Then push the finished feature to github and open a pull request form the branch.

4.1 New Release

To create a new release:

1. Add changes to `docs/source/changelog.rst`, using Releases formatting
2. Change version in `setup.py`
3. Change version in `docs/source/conf.py`
4. `python setup.py sdist upload`
5. `python setup.py bdist_wheel`

6. `git tag x.x.x`
7. Push git tag and commit
8. Add release to github tag, with changes and releasion name.

CHAPTER 5

Changelog

- : Added support for Django 1.9.
- #26: Removed lambdas in docs for model.
- : Reimplement progressive and optimize support.
- : Added option to not overwrite image.
- : Increased transform debug logging.
- : Save color profiles with images
- : Fixed Celery task defination.
- : Removed extranious *print* debugging.
- : Added support for Python 3.5 and Django 1.7, 1.8.
- : Added support for Celery.
- : Remove support for PQ (it isn't being maintained).
- : Changed to use Docker for development.
- #16: Support Django 1.7 (experimental).
- #15: Make compatible with `height_field` and `width_field`.
- #13: Add testing for 3rd party transformation support.
- #19: Change to use Releases for changelog.
- : Fix height/width order. Before they were reversed and broken.
- : Deletion of destination field when no source exists or transformation fails.
- : Require Pillow.
- : Fixed spelling for caller setting.
- : Display progress for management command.
- : Check if destination field exists before deleting.

- : Fixed adding management command directory
- : Added management directory to packages so that Django finds command
- : Added option to django-rq
- : Use py.test for testing.
- : Added Sphinx docs.
- : Save only filename and not whole path for transformed images.
- : Use .count() for management command instead of len()
- : Fixed `retransform` with no fields.
- : Add all packages so that Django finds management command
- : Zip safe on setup.py so Django finds management command.
- : Support uploading of non-image files.
- : Save image with higher quality.
- : Save image as progressive.
- #20: Don't save image with optimize either, because encoutner error.
- : Fixed Readme formatting.
- : Added requirement for at least Django 1.5.
- : Added instructions to add to `INSTALLED_APPS`.
- : Reasons why to use library added to readme.
- : Don't save image as progressive, because encounters error.
- : Convert image to JPEG colorspace.
- : Addressed force_update error.
- : Transform post save.
- : Moved error handling to transform function.
- : Make sure image exists before trying to delete it.
- : Basic functionality.

CHAPTER 6

simpleimages Package

6.1 callers Module

simpleimages.callers.**celery**(*function*, **args*, ***kwargs*)
Calls function asynchronously by creating a pickling it and calling it in a task.

simpleimages.callers.**default**(*function*, **args*, ***kwargs*)
Calls function with any passed in args and ``kwargs.

6.2 management.commands.retransform Module

simpleimages.management.commands.retransform.**parse_model_specifier**(*specifier*)
Parses a string that specifies either a model or a field. The string should look like app.model.[field].

```
>>> print parse_model_specifier('tests.TestModel')
(<class 'tests.models.TestModel'>, None)
>>> print parse_model_specifier('tests.TestModel.image')
(<class 'tests.models.TestModel'>, 'image')
```

Returns model and (optionally) field name

Return type tuple of Model and str or None

6.3 trackers Module

simpleimages.trackers.**track_model**(*model*)
Perform designated transformations on model, when it saves.

Calls *perform_transformation()* on every model saves using django.db.models.signals.post_save.

It uses the `update_fields` kwarg to tell what fields it should transform.

6.4 `transforms` Module

class `simpleimages.transforms.`**`BasePILTransform`**

Bases: `object`

Base transform object that provides helper methods to transform `django.core.files.images.ImageFile` using `PIL`.

Must subclass and override *`transform_pil_image()`*.

`__call__`(*original_django_file*)

Returns the transformed version of `PIL.Image.Image`

Uses *`transform_pil_image()`* to transform the `PIL.Image.Image`.

Parameters **`original_django_file`** (`django.core.files.images.ImageFile`) – source file

Returns transformed file

Return type `django.core.files.File`

`django_file_to_pil_image`(*django_file*)

Converts a the file returned by `django.db.models.fields.ImageField` to a PIL image.

Parameters **`django_file`** (`django.db.models.fields.files.FieldFile`) – django file

Return type `PIL.Image.Image`

`pil_image_to_django_file`(*pil_image*)

Gets a PIL image ready to be able to be saved using `django.db.models.fields.files.FieldFile.save()`

It converts the mode first to `RGB` or `L`, so that it can then save it as a `JPEG`. It will save it as a progressive `JPEG` with a quality of `IMAGE_QUALITY`.

Parameters **`pil_image`** (`PIL.Image.Image`) – original image

Returns transformed image

Return type `django.core.files.base.ContentFile`

`transform_pil_image`(*pil_image*)

Returns the transformed version of the `PIL.Image.Image` Do some logic on `PIL.Image.Image`.

Must subclass method to provide transformation logic.

Parameters **`pil_image`** (`PIL.Image.Image`) – original image

Returns transformed image

Return type `PIL.Image.Image`

class `simpleimages.transforms.`**`Scale`**(*width=None*, *height=None*)

Bases: *`simpleimages.transforms.BasePILTransform`*

Scales down an image to max height and/or width. If the original image is smaller than both/either specified dimensions than it will be left unchanged.

`__init__`(*width=None*, *height=None*)

Initialize this class with a max height and/or width (in pixels).

Parameters

- **height** (*int or float*) – max height of scaled image
- **width** (*int or float*) – max width of scaled image

transform_pil_image (*pil_image*)

Uses `PIL.Image.Image.transform()` to scale down the image.

Based on this stackoverflow discussions, uses `PIL.Image.ANTIALIAS`

6.5 `utils` Module

`simpleimages.utils.`**perform_transformation**(*instance*, *field_names_to_transform=None*)

Transforms a model based on the fields specified in the `transformed_fields` attribute. This should map source image field names to dictionaries mapping destination field name to their transformations. For instance:

```
{
    'image': {
        'thumbnail': scale(width=10),
    }
}
```

If `field_names_to_transform` is None, then it will transform all fields. Otherwise it will only transform from those fields specified in `field_names_to_transform`.

Parameters

- **instance** (instance of `django.db.models.Model`) – model instance to perform transformations on
- **field_names_to_transform** (*iterable of strings or None*) – field names on model to perform transformations on

`simpleimages.utils.`**transform_field**(*instance*, *source_field_name*, *destination_field_name*, *transformation*)

Does an image transformation on a instance. It will get the image from the source field attribute of the instnace, then call the transformation function with that instance, and finally save that transformed image into the destination field attribute of the instance.

Note: If the source field is blank or the transformation returns a false value then the destination field image will be deleted, if it exists.

Warning: When the model instance is saved with the new transformed image, it uses the `update_fields` argument for `save()`, to tell the model to only update the destination field and, if set in the destination field, the `height_field` and `width_field`. This means that if the saving code for the model sets any other fields, in the saving field process, it will not save those fields to the database. This would only happen if you introduce custom logic to the saving process of destination field, like the dimension fields do, that updates another field on that module. In that case, when the model is saved for the transformation, that other field will not be saved to the database.

Parameters

- **instance** (instance of `django.db.models.Model`) – model instance to perform transformations on

- **`source_field_name`** (*`string`*) – field name on model to find source image
- **`destination_field_name`** (*`string`*) – field name on model save transformed image to
- **`transformation`** (*`function`*) – function, such as `scale()`, that takes an image files and returns a transformed image

Python Module Index

Index