Django Daguerre Documentation

Release 3.0.0

Stephen Burrows, Harris Lapiroff

Apr 08, 2020

Contents

Django Daguerre manipulates images on the fly. Use it to scale images up or down. Use it to generate thumbnails in bulk or sets of responsive images without slowing down your templates. Or customize it to do even more powerful image processing.

You don't need to run a cron job ahead of time. You don't need to make any changes to your models. It **just works**.

```
{% load daguerre %}
<img src="{% adjust my_model.image "fill" width=200 height=400 %}" />

{% adjust_
↪bulk my_queryset "method.image" "fill" width=200 height=400 as adjusted_list %}
{% for my_model, image in adjusted_list %}
  <img src="{{ image }}" />
{% endfor %}
```

Code http://github.com/melinath/django-daguerre

Docs http://readthedocs.org/docs/django-daguerre/

Build status ![build passing]

Fig. 1: Louis Daguerre, Father of Photography

Contents

1.1 Installation and Setup

1.1.1 Installation

Install the latest version of Daguerre using `pip`:

```
pip install django-daguerre
```

You can also clone the repository or download a package at https://github.com/melinath/django-daguerre.

1.1.2 Setup

Add `'daguerre'` to your project's `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    'daguerre',
    ...
)
```

Add Daguerre's URL patterns to your URLconf:

```
urlpatterns = patterns('',
    url(r'^daguerre/', include('daguerre.urls')),
    ...
)
```

Run the migration command to create the database models:

```
python manage.py migrate daguerre
```

Now you're ready to *use Daguerre's template tags*!

1.1.3 Versions and Requirements

- Python 3.5+
- Pillow
- Django 1.11 – 3.0

Daguerre *may* work with earlier versions of these packages, but they are not officially supported.

1.2 Template Tags

1.2.1 adjust

The easiest way to use Daguerre is through the *{% adjust %}* template tag:

```
{% load daguerre %}
<img src="{% adjust my_model.image 'fill' width=128 height=256 %}" />
```

The *{% adjust %}* tag works directly with any ImageField (or storage path). There is no magic. You don't need to change your models. It Just Works.

Daguerre provides a number of built-in adjustments (such as 'fill') which can be used with the *{% adjust %}* out of the box, as well as an API for registering custom adjustments.

Take this picture:

Fig. 1: Full size: 800x600. This photograph, by Wikipedia user Sloesch, is licensed as CC BY-SA.

Let's use *{% adjust %}* with width 200 (25%) and height 300 (50%), with three of the built-in adjustments.

"fit"	"fill"	"crop"
Fits the entire image into the given dimensions without distorting it.	Fills the entire space given by the dimensions by cropping to the same width/height ratio and then scaling down or up.	Crops the image to the given dimensions without any resizing.

Chaining Adjustments

You can also use the *{% adjust %}* tag to chain multiple adjustments. Take the following:

```
{% load daguerre %}
{% adjust my_model.image 'ratiocrop' ratio='16:9' 'fit' width=200 %}
```

This tag first crops the image to a 16:9 ratio, then scales as much as needed to fit within a 200-pixel width. In other words:

See also:

daguerre.adjustments for more built-in adjustments.

Getting adjusted width and height

```
{% load daguerre %}
{% adjust my_model.image 'fit' width=128 height=128 as image %}
<img src="{{ image }}" width="{{ image.width }}" height="{{ image.height }}" />
```

The object being set to the `image` context variable is an `AdjustmentInfoDict` instance. In addition to rendering as the URL for an image, this object provides access to some other useful pieces of information—in particular, the width and height that the adjusted image *will have*, based on the width and height of the original image and the parameters given to the tag. This can help you avoid changes to page flow as adjusted images load.

Let's be lazy

So the `{% adjust %}` tag renders as a URL to adjusted image, right? Yes, but as lazily as possible. If the adjustment has already been performed, the adjusted image's URL is fetched from the database. If the adjustment has *not* been performed, the tag renders as a URL to a view which, when accessed, will create an adjusted version of the image and return a redirect to the adjusted image's actual URL.

This does have the downside of requiring an additional request/response cycle when unadjusted images are fetched by the user – but it has the upside that no matter how many `{% adjust %}` tags you have on a page, the initial load of the page won't be slowed down by (potentially numerous, potentially expensive) image adjustments.

Note: The adjustment view has some light security in place to make sure that users can't run arbitrary image resizes on your servers.

1.2.2 adjust_bulk

If you are using a large number of similar adjustments in one template - say, looping over a queryset and adjusting the same attribute each time - you can save yourself queries by using `{% adjust_bulk %}`.

```
{% load daguerre %}
{% adjust_bulk my_queryset "method.image" "fill" width=200 height=400 as adjusted_
→list %}
{% for my_model, image in adjusted_list %}
  <img src="{{ image }}" />
{% endfor %}
```

The syntax is similar to `{% adjust %}`, except that:

- `as <varname>` is required.
- an iterable (`my_queryset`) and a lookup to be performed on each item in the iterable (`"method.image"`) are provided in place of an image file or storage path. (If the iterable is an iterable of image files or storage paths, the lookup is not required.)

You've got everything you need now to use Daguerre and resize images like a champ. But what if you need more control over *how* your images are cropped? Read on to learn about *Smart Cropping with Areas*.

1.3 Smart Cropping with Areas

Daguerre allows you to influence how images are cropped with *Areas*.

1.3.1 Use the AreaWidget

Daguerre provides a widget which can be used with any `ImageField` to edit *Areas* for that image file. Add this formfield override to your ModelAdmin to enable the widget.

```
from daguerre.widgets import AreaWidget

class YourModelAdmin(admin.ModelAdmin):
    formfield_overrides = {
        models.ImageField: {'widget': AreaWidget},
    }
    ...
```
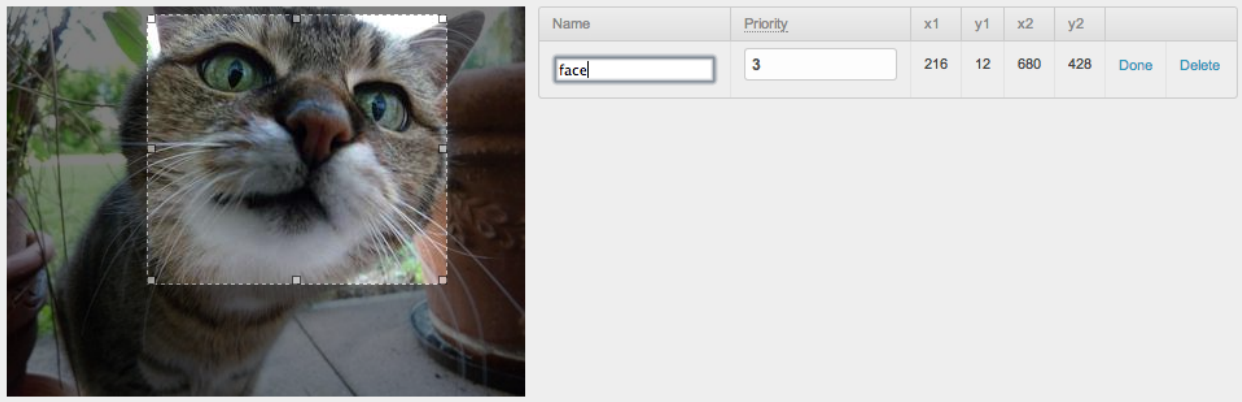

Fig. 2: The `AreaWidget` allows you to define areas of an image with click-and-drag. (Screenshot includes Grappelli.)

1.3.2 Adjustments with Areas

After you define *Areas* for an image in the admin, adjustments that remove parts of the image (such as crop) will protect those parts of the image during processing. See the difference in this adjustment.
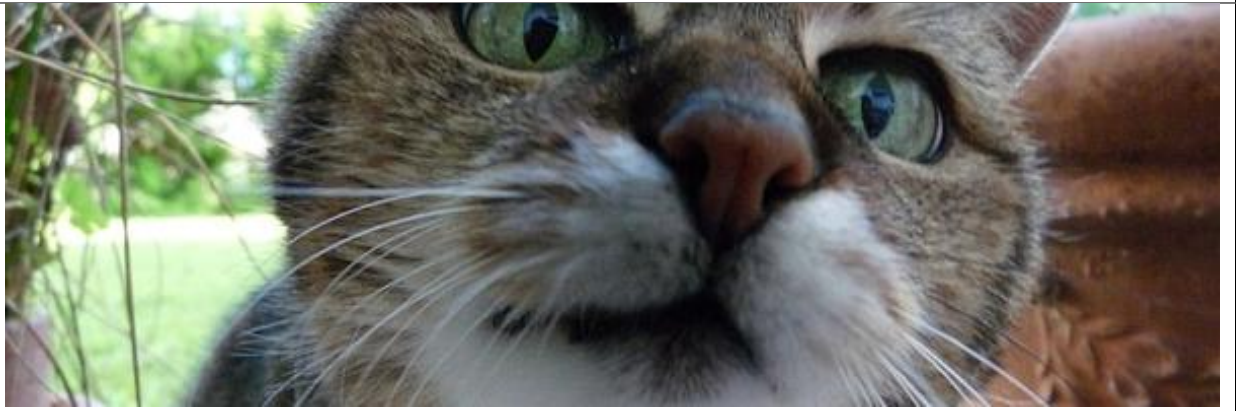
```
<img src="{% adjust my_model.image "fill" width=600 height=200 %}" />
```


Result without 'face' Area defined

Result with 'face' Area defined

1.3.3 Areas and namedcrop

You can also use the built-in "namedcrop" adjustment force a specific crop.

```html
<img src="{% adjust my_model.image "namedcrop" name="face" %}" />
```

1.4 Management Commands

1.4.1 `./manage.py daguerre clean`

Cleans out extra or invalid data stored by daguerre:

- *AdjustedImages* and *Areas* that reference storage paths which no longer exist.
- Duplicate *AdjustedImages*.
- Adjusted image files which don't have an associated *AdjustedImage*.
- *AdjustedImage* instances with missing adjusted image files.

1.4.2 `./manage.py daguerre preadjust [--remove] [--nocreate]`

Looks for a `DAGUERRE_PREADJUSTMENTS` setting using the following structure:

```python
from daguerre.adjustments import Fit, Fill
DAGUERRE_PREADJUSTMENTS = (
    ('myapp.MyModel',
     [Fit(width=800, height=500)],
     'template.style.lookup'),
    (OtherModel.objects.filter(field=value),
     [Fill(width=300, height=216)].
     'template.style.lookup'),
    ...
)
```

Essentially, this is expected to be an iterable of tuples, where each tuple contains three items:

1. `'<applabel>.<model>'`, a model class, a queryset, or any iterable.

2. A non-empty iterable of adjustment instances to be applied to each image.

3. A template-style lookup (or None).

Each time the command is run, the first item will be used to generate a fresh iterable of model instances. The lookup will be applied to each instance to get an `ImageFile` or storage path, which will then have the list of adjustments applied it to create a new adjusted version of the image, if one doesn't exist already. (This is essentially the same functionality as the `{% adjust_bulk %}` template tag.)

If `--remove` is specified, the command will delete all `AdjustedImage` instances which would not be generated by the parameters specified in DAGUERRE_PREADJUSTMENTS.

If `--nocreate` is specified, the command will not create any new `AdjustedImage` instances. This can be used with `--remove` to just prune instances that aren't specified in DAGUERRE_PREADJUSTMENTS without creating new pre-adjusted instances. Specifying `--nocreate` *without* `--remove` makes this command a no-op.

1.5 Custom settings

1.5.1 Adjust the image path

The variations are stored under a hashed directory path that starts with the `dg` directory by default (e.g. `dg/ce/2b/7014c0bdbedea0e4f4bf.jpeg`). This setting can be modified in the project's settings by setting the `DAGUERRE_ADJUSTED_IMAGE_PATH` variable.

Example:

```python
# settings.py
DAGUERRE_ADJUSTED_IMAGE_PATH = 'img'
```

which would produce the following path: `img/ce/2b/7014c0bdbedea0e4f4bf.jpeg`

> **Warning:** The maximum length of the `DAGUERRE_ADJUSTED_IMAGE_PATH` string is 13 characters. If the string has more than 13 characters, it will gracefully fall back to the the default value, i.e. `dg`

API Docs

2.1 Adjustments

Daguerre provides a variety of adjustments to use when processing images, as well as an API for registering custom adjustments.

class `daguerre.adjustments.`**`Adjustment`**(***kwargs*)

> Base class for all adjustments which can be carried out on an image. The adjustment itself represents a set of parameters, which can then be applied to images (taking areas into account if applicable).
>
> Adjustment subclasses need to define two methods: *calculate()* and *adjust()*. If the method doesn't use areas, you can set the `uses_areas` attribute on the method to `False` to optimize adjustment.
>
> > **Parameters kwargs** – The requested kwargs for the adjustment. The keys must be in *parameters* or the adjustment is invalid.
>
> **`adjust`**(*image*, *areas=None*)
>
> > Manipulates and returns the image. Must be implemented by subclasses.
> >
> > > **Parameters**
> > >
> > > - **`image`** – PIL Image which will be adjusted.
> > >
> > > - **`areas`** – iterable of *Area* instances to be considered in performing the adjustment.
>
> **`calculate`**(*dims*, *areas=None*)
>
> > Calculates the dimensions of the adjusted image without actually manipulating the image. By default, just returns the given dimensions.
> >
> > > **Parameters**
> > >
> > > - **`dims`** – (`width, height`) tuple of the current image dimensions.
> > >
> > > - **`areas`** – iterable of *Area* instances to be considered in calculating the adjustment.
>
> **`parameters = ()`**
>
> > Accepted parameters for this adjustment - for example, `"width"`, `"height"`, `"color"`, `"unicorns"`, etc.

2.1.1 Built-In Adjustments

class daguerre.adjustments.**Fit**(***kwargs*)

> Resizes an image to fit entirely within the given dimensions without cropping and maintaining the width/height ratio.
>
> If neither width nor height is specified, this adjustment will simply return a copy of the image.
>
> **parameters = ('width', 'height')**

class daguerre.adjustments.**Fill**(***kwargs*)

> Crops the image to the requested ratio (using the same logic as *Crop* to protect *Area* instances which are passed in), then resizes it to the actual requested dimensions. If width or height is not given, then the unspecified dimension will be allowed to expand up to max_width or max_height, respectively.
>
> **parameters = ('width', 'height', 'max_width', 'max_height')**

class daguerre.adjustments.**Crop**(***kwargs*)

> Crops an image to the given width and height, without scaling it. *Area* instances which are passed in will be protected as much as possible during the crop.
>
> **parameters = ('width', 'height')**

class daguerre.adjustments.**RatioCrop**(***kwargs*)

> Crops an image to the given aspect ratio, without scaling it. *Area* instances which are passed in will be protected as much as possible during the crop.
>
> **parameters = ('ratio',)**
> > ratio should be formatted as "<width>:<height>"

class daguerre.adjustments.**NamedCrop**(***kwargs*)

> Crops an image to the given named area, without scaling it. *Area* instances which are passed in will be protected as much as possible during the crop.
>
> If no area with the given name exists, this adjustment is a no-op.
>
> **parameters = ('name',)**

When used with the template tag, these adjustments should be referred to by their lowercase name:

```
{% adjust image "fit" width=300 %}
```

See *Template Tags* for examples.

2.1.2 Custom Adjustments

You can easily add custom adjustments for your particular project. For example, an adjustment to make an image grayscale might look something like this:

```
# Somewhere that will be imported.
from daguerre.adjustments import Adjustment, registry
from PIL import ImageOps

@registry.register
class GrayScale(Adjustment):
    def adjust(self, image, areas=None):
        return ImageOps.grayscale(image)
    adjust.uses_areas = False
```

Now you can use your adjustment in templates:

```
{% adjust image "grayscale" %}
```

2.2 Models

class daguerre.models.**AdjustedImage**(**args*, ***kwargs*)

Represents a managed image adjustment.

> **exception DoesNotExist**
>
> **exception MultipleObjectsReturned**

class daguerre.models.**Area**(**args*, ***kwargs*)

Represents an area of an image. Can be used to specify a crop. Also used for priority-aware automated image cropping.

> **exception DoesNotExist**
>
> **exception MultipleObjectsReturned**
>
> **clean**()
>
> > Hook for doing any extra model-wide validation after clean() has been called on every field by self.clean_fields. Any ValidationError raised by this method will not be associated with a particular field; it will have a special-case association with the field defined by NON_FIELD_ERRORS.
>
> **clean_fields**(*exclude=None*)
>
> > Clean all fields and raise a ValidationError containing a dict of all validation errors if any occur.

```
{% adjust image "grayscale" %}
```

Release Notes

3.1 Release Notes

3.1.1 3.0.0 (2020-04-08)

- Added official support for Django 3.0
- Added official support for Python 3.8
- Dropped support for Django < 1.11
- Dropped support for Python 2
- Dropped support for PIL

3.1.2 2.3.1 (2020-04-06)

- Bugfix: Made {% adjust ... as var %} template tag correctly use an info dict instead of a helper instance so that width and height are available. Thanks @pembeci for the report!

3.1.3 2.3.0 (2020-04-02)

- Added official support for Django 2.0-2.2
- Dropped official support for Python 3.3 and Python 3.4
- Updated get_image_dimensions implementation to better match current Django implementation

3.1.4 2.2.3 (2017-04-27)

- Doc updates
- Corrected preadjust behavior

- Corrected rounding of floating point numbers (thanks @fitodic!)
- Allowed customizing the image storage path, and improved the default path (thanks @fitodic!)
- Added official support for Django 1.10 & 1.11

3.1.5 2.2.2 (2016-05-10)

- Fixed incorrect requirements in setup.py.
- Fixed tests for adjusting broken images so they're compatible with all versions of Pillow.

3.1.6 2.2.1 (2016-05-05)

- Added compatibility with Django 1.9 (thanks @ajhi!)

3.1.7 2.2.0 (2016-04-13)

- Improved efficiency of adjusted image queries (thanks @mislavcimpersak!)
- Fixed some minor image adjustment bugs.
- Added Jinja template support.
- Added compatibility with Django 1.8

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

d