ImageKit Documentation

Release 3.2.6

Justin Driscoll, Bryan Veloso, Greg Newman, Chris Drackett & Ma

July 26, 2015

Contents

ImageKit is a Django app for processing images. Need a thumbnail? A black-and-white version of a user-uploaded image? ImageKit will make them for you. If you need to programatically generate one image from another, you need ImageKit.

ImageKit comes with a bunch of image processors for common tasks like resizing and cropping, but you can also create your own. For an idea of what's possible, check out the Instakit project.

For the complete documentation on the latest stable version of ImageKit, see ImageKit on RTD.

1. Install PIL or Pillow. (If you're using an `ImageField` in Django, you should have already done this.)
2. `pip install django-imagekit`
3. Add `'imagekit'` to your `INSTALLED_APPS` list in your project's settings.py

Note: If you've never seen Pillow before, it considers itself a more-frequently updated "friendly" fork of PIL that's compatible with setuptools. As such, it shares the same namespace as PIL does and is a drop-in replacement.

CHAPTER 1

Usage Overview

1.1 Specs

You have one image and you want to do something to it to create another image. But how do you tell ImageKit what to do? By defining an image spec.

An **image spec** is a type of **image generator** that generates a new image from a source image.

1.1.1 Defining Specs In Models

The easiest way to use define an image spec is by using an ImageSpecField on your model class:

```
from django.db import models
from imagekit.models import ImageSpecField
from imagekit.processors import ResizeToFill

class Profile(models.Model):
    avatar = models.ImageField(upload_to='avatars')
    avatar_thumbnail = ImageSpecField(source='avatar',
                                      processors=[ResizeToFill(100, 50)],
                                      format='JPEG',
                                      options={'quality': 60})

profile = Profile.objects.all()[0]
print profile.avatar_thumbnail.url    # > /media/CACHE/images/982d5af84cddddfd0fbf70892b4431e4.jpg
print profile.avatar_thumbnail.width  # > 100
```

As you can probably tell, ImageSpecFields work a lot like Django's ImageFields. The difference is that they're automatically generated by ImageKit based on the instructions you give. In the example above, the avatar thumbnail is a resized version of the avatar image, saved as a JPEG with a quality of 60.

Sometimes, however, you don't need to keep the original image (the avatar in the above example); when the user uploads an image, you just want to process it and save the result. In those cases, you can use the `ProcessedImageField` class:

```
from django.db import models
from imagekit.models import ProcessedImageField

class Profile(models.Model):
    avatar_thumbnail = ProcessedImageField(upload_to='avatars',
                                           processors=[ResizeToFill(100, 50)],
                                           format='JPEG',
                                                    options={'quality': 60})

profile = Profile.objects.all()[0]
print profile.avatar_thumbnail.url    # > /media/avatars/MY-avatar.jpg
print profile.avatar_thumbnail.width  # > 100
```

This is pretty similar to our previous example. We don't need to specify a "source" any more since we're not processing another image field, but we do need to pass an "upload_to" argument. This behaves exactly as it does for Django ImageFields.

Note: You might be wondering why we didn't need an "upload_to" argument for our ImageSpecField. The reason is that ProcessedImageFields really are just like ImageFields—they save the file path in the database and you need to run syncdb (or create a migration) when you add one to your model.

ImageSpecFields, on the other hand, are virtual—they add no fields to your database and don't require a database. This is handy for a lot of reasons, but it means that the path to the image file needs to be programmatically constructed based on the source image and the spec.

1.1.2 Defining Specs Outside of Models

Defining specs as models fields is one very convenient way to process images, but it isn't the only way. Sometimes you can't (or don't want to) add fields to your models, and that's okay. You can define image spec classes and use them directly. This can be especially useful for doing image processing in views— particularly when the processing being done depends on user input.

```
from imagekit import ImageSpec
from imagekit.processors import ResizeToFill

class Thumbnail(ImageSpec):
    processors = [ResizeToFill(100, 50)]
    format = 'JPEG'
    options = {'quality': 60}
```

It's probably not surprising that this class is capable of processing an image in the exact same way as our ImageSpec-Field above. However, unlike with the image spec model field, this class doesn't define what source the spec is acting on, or what should be done with the result; that's up to you:

```
source_file = open('/path/to/myimage.jpg')
image_generator = Thumbnail(source=source_file)
result = image_generator.generate()
```

Note: You don't have to use `open`! You can use whatever File-like object you want—including a model's `ImageField`.

The result of calling `generate()` on an image spec is a file-like object containing our resized image, with which you can do whatever you want. For example, if you wanted to save it to disk:

```
dest = open('/path/to/dest.jpg', 'w')
dest.write(result.read())
dest.close()
```

1.1.3 Using Specs In Templates

If you have a model with an ImageSpecField or ProcessedImageField, you can easily use those processed image just as you would a normal image field:

```
<img src="{{ profile.avatar_thumbnail.url }}" />
```

(This is assuming you have a view that's setting a context variable named "profile" to an instance of our Profile model.)

But you can also generate processed image files directly in your template—from any image—without adding anything to your model. In order to do this, you'll first have to define an image generator class (remember, specs are a type of generator) in your app somewhere, just as we did in the last section. You'll also need a way of referring to the generator in your template, so you'll need to register it.

```
from imagekit import ImageSpec, register
from imagekit.processors import ResizeToFill

class Thumbnail(ImageSpec):
    processors = [ResizeToFill(100, 50)]
    format = 'JPEG'
    options = {'quality': 60}

register.generator('myapp:thumbnail', Thumbnail)
```

Note: You can register your generator with any id you want, but choose wisely! If you pick something too generic, you could have a conflict with another third-party app you're using. For this reason, it's a good idea to prefix your generator ids with the name of your app. Also, ImageKit recognizes colons as separators when doing pattern matching (e.g. in the generateimages management command), so it's a good idea to use those too!

Warning: This code can go in any file you want—but you need to make sure it's loaded! In order to keep things simple, ImageKit will automatically try to load an module named "imagegenerators" in each of your installed apps. So why don't you just save yourself the headache and put your image specs in there?

Now that we've created an image generator class and registered it with ImageKit, we can use it in our templates!

generateimage

The most generic template tag that ImageKit gives you is called "generateimage". It requires at least one argument: the id of a registered image generator. Additional keyword-style arguments are passed to the registered generator class. As we saw above, image spec constructors expect a source keyword argument, so that's what we need to pass to use our thumbnail spec:

```
{% load imagekit %}

{% generateimage 'myapp:thumbnail' source=source_file %}
```

This will output the following HTML:

```
<img src="/media/CACHE/images/982d5af84cddddfd0fbf70892b4431e4.jpg" width="100" height="50" />
```

You can also add additional HTML attributes; just separate them from your keyword args using two dashes:

```
{% load imagekit %}

{% generateimage 'myapp:thumbnail' source=source_file -- alt="A picture of Me" id="mypicture" %}
```

Not generating HTML image tags? No problem. The tag also functions as an assignment tag, providing access to the underlying file object:

```
{% load imagekit %}

{% generateimage 'myapp:thumbnail' source=source_file as th %}
<a href="{{ th.url }}">Click to download a cool {{ th.width }} x {{ th.height }} image!</a>
```

thumbnail

Because it's such a common use case, ImageKit also provides a "thumbnail" template tag:

```
{% load imagekit %}

{% thumbnail '100x50' source_file %}
```

Like the generateimage tag, the thumbnail tag outputs an <img> tag:

```
<img src="/media/CACHE/images/982d5af84cddddfd0fbf70892b4431e4.jpg" width="100" height="50" />
```

Comparing this syntax to the generateimage tag above, you'll notice a few differences.

First, we didn't have to specify an image generator id; unless we tell it otherwise, thumbnail tag uses the generator registered with the id "imagekit:thumbnail". **It's important to note that this tag is *not* using the Thumbnail spec class we defined earlier**; it's using the generator registered with the id "imagekit:thumbnail" which, by default, is `imagekit.generatorlibrary.Thumbnail`.

Second, we're passing two positional arguments (the dimensions and the source image) as opposed to the keyword arguments we used with the generateimage tag.

Like with the generateimage tag, you can also specify additional HTML attributes for the thumbnail tag, or use it as an assignment tag:

```
{% load imagekit %}

{% thumbnail '100x50' source_file -- alt="A picture of Me" id="mypicture" %}
{% thumbnail '100x50' source_file as th %}
```

1.1.4 Using Specs in Forms

In addition to the model field above, there's also a form field version of the `ProcessedImageField` class. The functionality is basically the same (it processes an image once and saves the result), but it's used in a form class:

```
from django import forms
from imagekit.forms import ProcessedImageField
from imagekit.processors import ResizeToFill

class ProfileForm(forms.Form):
    avatar_thumbnail = ProcessedImageField(spec_id='myapp:profile:avatar_thumbnail',
                                           processors=[ResizeToFill(100, 50)],
                                           format='JPEG',
                                           options={'quality': 60})
```

The benefit of using `imagekit.forms.ProcessedImageField` (as opposed to `imagekit.models.ProcessedImageField` above) is that it keeps the logic for creating the image outside of your model (in which you would use a normal Django ImageField). You can even create multiple forms, each with their own ProcessedImageField, that all store their results in the same image field.

1.2 Processors

So far, we've only seen one processor: `imagekit.processors.ResizeToFill`. But ImageKit is capable of far more than just resizing images, and that power comes from its processors.

Processors take a PIL image object, do something to it, and return a new one. A spec can make use of as many processors as you'd like, which will all be run in order.

```
from imagekit import ImageSpec
from imagekit.processors import TrimBorderColor, Adjust

class MySpec(ImageSpec):
    processors = [
        TrimBorderColor(),
        Adjust(contrast=1.2, sharpness=1.1),
    ]
    format = 'JPEG'
    options = {'quality': 60}
```

The `imagekit.processors` module contains processors for many common image manipulations, like resizing, rotating, and color adjustments. However, if they aren't up to the task, you can create your own. All you have to do is define a class that implements a `process()` method:

```
class Watermark(object):
    def process(self, image):
        # Code for adding the watermark goes here.
        return image
```

That's all there is to it! To use your fancy new custom processor, just include it in your spec's `processors` list:

```
from imagekit import ImageSpec
from imagekit.processors import TrimBorderColor, Adjust
from myapp.processors import Watermark

class MySpec(ImageSpec):
    processors = [
        TrimBorderColor(),
        Adjust(contrast=1.2, sharpness=1.1),
        Watermark(),
    ]
    format = 'JPEG'
    options = {'quality': 60}
```

Note that when you import a processor from `imagekit.processors`, imagekit in turn imports the processor from PILKit. So if you are looking for available processors, look at PILKit.

1.3 Admin

ImageKit also contains a class named `imagekit.admin.AdminThumbnail` for displaying specs (or even regular ImageFields) in the Django admin change list. AdminThumbnail is used as a property on Django admin classes:

```
from django.contrib import admin
from imagekit.admin import AdminThumbnail
from .models import Photo

class PhotoAdmin(admin.ModelAdmin):
    list_display = ('__str__', 'admin_thumbnail')
    admin_thumbnail = AdminThumbnail(image_field='thumbnail')

admin.site.register(Photo, PhotoAdmin)
```

AdminThumbnail can even use a custom template. For more information, see `imagekit.admin.AdminThumbnail`.

1.4 Management Commands

ImageKit has one management command—*generateimages*—which will generate cache files for all of your registered image generators. You can also pass it a list of generator ids in order to generate images selectively.

CHAPTER 2

Community

Please use the GitHub issue tracker to report bugs with django-imagekit. A mailing list also exists to discuss the project and ask questions, as well as the official #imagekit channel on Freenode.

Contributing

We love contributions! And you don't have to be an expert with the library—or even Django—to contribute either: ImageKit's processors are standalone classes that are completely separate from the more intimidating internals of Django's ORM. If you've written a processor that you think might be useful to other people, open a pull request so we can take a look!

You can also check out our list of open, contributor-friendly issues for ideas.

Check out our contributing guidelines for more information about pitching in with ImageKit.

CHAPTER 4

Authors

ImageKit was originally written by Justin Driscoll.

The field-based API and other post-1.0 stuff was written by the bright people at HZDG.

4.1 Maintainers

- Matthew Tretter
- Bryan Veloso
- Chris Drackett
- Greg Newman

4.2 Contributors

- Josh Ourisman
- Jonathan Slenders
- Eric Eldredge
- Chris McKenzie
- Markus Kaiserswerth
- Ryan Bagwell
- Alexander Bohn
- Timothée Peignier
- Madis Väin
- Jan Sagemüller
- Clay McClure
- Jannis Leidel
- Sean Bell
- Saul Shanabrook
- Venelin Stoykov

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

5.1 Configuration

5.1.1 Settings

`django.conf.settings.`**`IMAGEKIT_CACHEFILE_DIR`**

> **Default** `'CACHE/images'`

The directory to which image files will be cached.

`django.conf.settings.`**`IMAGEKIT_DEFAULT_FILE_STORAGE`**

> **Default** `None`

The qualified class name of a Django storage backend to use to save the cached images. If no value is provided for `IMAGEKIT_DEFAULT_FILE_STORAGE`, and none is specified by the spec definition, your default file storage will be used.

`django.conf.settings.`**`IMAGEKIT_DEFAULT_CACHEFILE_BACKEND`**

> **Default** `'imagekit.cachefiles.backends.Simple'`

Specifies the class that will be used to validate cached image files.

`django.conf.settings.`**`IMAGEKIT_DEFAULT_CACHEFILE_STRATEGY`**

> **Default** `'imagekit.cachefiles.strategies.JustInTime'`

The class responsible for specifying how and when cache files are generated.

`django.conf.settings.`**`IMAGEKIT_CACHE_BACKEND`**

> **Default** If `DEBUG` is `True`, `'django.core.cache.backends.dummy.DummyCache'`.

Otherwise, `'default'`.

The Django cache backend to be used to store information like the state of cached images (i.e. validated or not).

`django.conf.settings.`**`IMAGEKIT_CACHE_PREFIX`**

> **Default** `'imagekit:'`

A cache prefix to be used when values are stored in `IMAGEKIT_CACHE_BACKEND`

`django.conf.settings.`**`IMAGEKIT_CACHEFILE_NAMER`**

Default `'imagekit.cachefiles.namers.hash'`

A function responsible for generating file names for non-spec cache files.

`django.conf.settings.`**`IMAGEKIT_SPEC_CACHEFILE_NAMER`**

Default `'imagekit.cachefiles.namers.source_name_as_path'`

A function responsible for generating file names for cache files that correspond to image specs. Since you will likely want to base the name of your cache files on the name of the source, this extra setting is provided.

5.2 Advanced Usage

5.2.1 Models

The `ImageSpecField` Shorthand Syntax

If you've read the README, you already know what an `ImageSpecField` is and the basics of defining one:

```
from django.db import models
from imagekit.models import ImageSpecField
from imagekit.processors import ResizeToFill

class Profile(models.Model):
    avatar = models.ImageField(upload_to='avatars')
    avatar_thumbnail = ImageSpecField(source='avatar',
                                      processors=[ResizeToFill(100, 50)],
                                      format='JPEG',
                                      options={'quality': 60})
```

This will create an `avatar_thumbnail` field which is a resized version of the image stored in the `avatar` image field. But this is actually just shorthand for creating an `ImageSpec`, registering it, and associating it with an `ImageSpecField`:

```
from django.db import models
from imagekit import ImageSpec, register
from imagekit.models import ImageSpecField
from imagekit.processors import ResizeToFill

class AvatarThumbnail(ImageSpec):
    processors = [ResizeToFill(100, 50)]
    format = 'JPEG'
    options = {'quality': 60}

register.generator('myapp:profile:avatar_thumbnail', AvatarThumbnail)

class Profile(models.Model):
    avatar = models.ImageField(upload_to='avatars')
    avatar_thumbnail = ImageSpecField(source='avatar',
                                      id='myapp:profile:avatar_thumbnail')
```

Obviously, the shorthand version is a lot, well…shorter. So why would you ever want to go through the trouble of using the long form? The answer is that the long form—creating an image spec class and registering it—gives you a lot more power over the generated image.

Specs That Change

As you'll remember from the README, an image spec is just a type of image generator that generates a new image from a source image. How does the image spec get access to the source image? Simple! It's passed to the constructor as a keyword argument and stored as an attribute of the spec. Normally, we don't have to concern ourselves with this; the `ImageSpec` knows what to do with the source image and we're happy to let it do its thing. However, having access to the source image in our spec class can be very useful…

Often, when using an `ImageSpecField`, you may want the spec to vary based on properties of a model. (For example, you might want to store image dimensions on the model and then use them to generate your thumbnail.) Now that we know how to access the source image from our spec, it's a simple matter to extract its model and use it to create our processors list. In fact, ImageKit includes a utility for getting this information.

```
from django.db import models
from imagekit import ImageSpec, register
from imagekit.models import ImageSpecField
from imagekit.processors import ResizeToFill
from imagekit.utils import get_field_info

class AvatarThumbnail(ImageSpec):
    format = 'JPEG'
    options = {'quality': 60}

    @property
    def processors(self):
        model, field_name = get_field_info(self.source)
        return [ResizeToFill(model.thumbnail_width, thumbnail.avatar_height)]

register.generator('myapp:profile:avatar_thumbnail', AvatarThumbnail)

class Profile(models.Model):
    avatar = models.ImageField(upload_to='avatars')
    avatar_thumbnail = ImageSpecField(source='avatar',
                                      id='myapp:profile:avatar_thumbnail')
    thumbnail_width = models.PositiveIntegerField()
    thumbnail_height = models.PositiveIntegerField()
```

Now each avatar thumbnail will be resized according to the dimensions stored on the model!

Of course, processors aren't the only thing that can vary based on the model of the source image; spec behavior can change in any way you want.

5.2.2 Source Groups

When you run the `generateimages` management command, how does ImageKit know which source images to use with which specs? Obviously, when you define an ImageSpecField, the source image is being connected to a spec, but what's going on underneath the hood?

The answer is that, when you define an ImageSpecField, ImageKit automatically creates and registers an object called a *source group*. Source groups are responsible for two things:

1. They dispatch signals when a source is saved, and
2. They expose a generator method that enumerates source files.

When these objects are registered (using `imagekit.register.source_group()`), their signals will trigger callbacks on the cache file strategies associated with image specs that use the source. (So, for example, you can chose to generate a file every time the source image changes.) In addition, the generator method is used (indirectly) to create the list of files to generate with the `generateimages` management command.

Currently, there is only one source group class bundled with ImageKit—the one used by ImageSpec-Fields. This source group (`imagekit.specs.sourcegroups.ImageFieldSourceGroup`) represents an ImageField on every instance of a particular model. In terms of the above description, the instance `ImageFieldSourceGroup(Profile, 'avatar')` 1) dispatches a signal every time the image in Profile's avatar ImageField changes, and 2) exposes a generator method that iterates over every Profile's "avatar" image.

Chances are, this is the only source group you will ever need to use, however, ImageKit lets you define and register custom source groups easily. This may be useful, for example, if you're using the template tags "generateimage" and "thumbnail" and the optimistic cache file strategy. Again, the purpose is to tell ImageKit which specs are used with which sources (so the "generateimages" management command can generate those files) and when the source image has been created or changed (so that the strategy has the opportunity to act on it).

A simple example of a custom source group class is as follows:

```
import glob
import os

class JpegsInADirectory(object):
    def __init__(self, dir):
        self.dir = dir

    def files(self):
        os.chdir(self.dir)
        for name in glob.glob('*.jpg'):
            yield open(name)
```

Instances of this class could then be registered with one or more spec id:

```
from imagekit import register

register.source_group('myapp:profile:avatar_thumbnail', JpegsInADirectory('/path/to/some/pics'))
```

Running the "generateimages" management command would now cause thumbnails to be generated (using the "myapp:profile:avatar_thumbnail" spec) for each of the JPEGs in */path/to/some/pics*.

Note that, since this source group doesnt send the *source_saved* signal, the corresponding cache file strategy callbacks would not be called for them.

5.3 Caching

5.3.1 Default Backend Workflow

`ImageSpec`

At the heart of ImageKit are image generators. These are classes with a `generate()` method which returns an image file. An image spec is a type of image generator. The thing that makes specs special is that they accept a source image. So an image spec is just an image generator that makes an image from some other image.

`ImageCacheFile`

However, an image spec by itself would be vastly inefficient. Every time an an image was accessed in some way, it would have be regenerated and saved. Most of the time, you want to re-use a previously generated image, based on the input image and spec, instead of generating a new one. That's where ImageCacheFile comes in. ImageCacheFile is a File-like object that wraps an image generator. They look and feel just like regular file objects, but they've got a little trick up their sleeve: they represent files that may not actually exist!

Cache File Strategy

Each ImageCacheFile has a cache file strategy, which abstracts away when image is actually generated. It can implement the following three methods:

- on_content_required - called by ImageCacheFile when it requires the contents of the generated image. For example, when you call read() or try to access information contained in the file.
- on_existence_required - called by ImageCacheFile when it requires the generated image to exist but may not be concerned with its contents. For example, when you access its url or path attribute.
- on_source_saved - called when the source of a spec is saved

The default strategy only defines the first two of these, as follows:

```
class JustInTime(object):
    def on_content_required(self, file):
        file.generate()

    def on_existence_required(self, file):
        file.generate()
```

Cache File Backend

The generate method on the ImageCacheFile is further delegated to the cache file backend, which abstracts away how an image is generated.

The cache file backend defaults to the setting IMAGEKIT_DEFAULT_CACHEFILE_BACKEND and can be set explicitly on a spec with the cachefile_backend attribute.

The default works like this:

- **Checks the file storage to see if a file exists**
    - If not, caches that information for 5 seconds
    - If it does, caches that information in the IMAGEKIT_CACHE_BACKEND

If file doesn't exist, generates it immediately and synchronously

That pretty much covers the architecture of the caching layer, and its default behavior. I like the default behavior. When will an image be regenerated? Whenever it needs to be! When will your storage backend get hit? Depending on your IMAGEKIT_CACHE_BACKEND settings, as little as twice per file (once for the existence check and once to save the generated file). What if you want to change a spec? The generated file name (which is used as part of the cache keys) vary with the source file name and spec attributes, so if you change any of those, a new file will be generated. The default behavior is easy!

Note: Like regular Django ImageFields, IK doesn't currently cache width and height values, so accessing those will always result in a read. That will probably change soon though.

5.3.2 Optimizing

There are several ways to improve the performance (reduce I/O operations) of ImageKit. Each has its own pros and cons.

Caching Data About Generated Files

The easiest, and most significant improvement you can make to improve the performance of your site is to have ImageKit cache the state of your generated files. The default cache file backend will already do this (if `DEBUG` is `False`), using your default Django cache backend, but you can make it way better by setting `IMAGEKIT_CACHE_BACKEND`. Generally, once a file is generated, you will never be removing it; therefore, if you can, you should set `IMAGEKIT_CACHE_BACKEND` to a cache backend that will cache forever.

Pre-Generating Images

The default cache file backend generates images immediately and synchronously. If you don't do anything special, that will be when they are first requested—as part of request-response cycle. This means that the first visitor to your page will have to wait for the file to be created before they see any HTML.

This can be mitigated, though, by simply generating the images ahead of time, by running the `generateimages` management command.

Note: If using with template tags, be sure to read *Source Groups*.

Deferring Image Generation

As mentioned above, image generation is normally done synchronously. through the default cache file backend. However, you can also take advantage of deferred generation. In order to do this, you'll need to do two things:

1. install celery (or django-celery if you are bound to Celery<3.1)
2. tell ImageKit to use the async cachefile backend. To do this for all specs, set the `IMAGEKIT_DEFAULT_CACHEFILE_BACKEND` in your settings

```
IMAGEKIT_DEFAULT_CACHEFILE_BACKEND = 'imagekit.cachefiles.backends.Async'
```

Images will now be generated asynchronously. But watch out! Asynchrounous generation means you'll have to account for images that haven't been generated yet. You can do this by checking the truthiness of your files; if an image hasn't been generated, it will be falsy:

```
{% if not profile.avatar_thumbnail %}
    <img src="/path/to/placeholder.jpg" />
{% else %}
    <img src="{{ profile.avatar_thumbnail.url }}" />
{% endif %}
```

Or, in Python:

```
profile = Profile.objects.all()[0]
if profile.avatar_thumbnail:
    url = profile.avatar_thumbnail.url
else:
    url = '/path/to/placeholder.jpg'
```

Note: If you are using an "async" backend in combination with the "optimistic" cache file strategy (see *Removing Safeguards* below), checking for thruthiness as described above will not work. The "optimistic" backend is very optimistic so to say, and removes the check. Create and use the following strategy to a) have images only created on save, and b) retain the ability to check whether the images have already been created:

```
class ImagekitOnSaveStrategy(object):
    def on_source_saved(self, file):
        file.generate()
```

Removing Safeguards

Even with pre-generating images, ImageKit will still try to ensure that your image exists when you access it by default. This is for your benefit: if you forget to generate your images, ImageKit will see that and generate it for you. If the state of the file is cached (see above), this is a pretty cheap operation. However, if the state isn't cached, ImageKit will need to query the storage backend.

For those who aren't willing to accept that cost (and who never want ImageKit to generate images in the request-responce cycle), there's the "optimistic" cache file strategy. This strategy only generates a new image when a spec's source image is created or changed. Unlike with the "just in time" strategy, accessing the file won't cause it to be generated, ImageKit will just assume that it already exists.

To use this cache file strategy for all specs, set the `IMAGEKIT_DEFAULT_CACHEFILE_STRATEGY` in your settings:

```
IMAGEKIT_DEFAULT_CACHEFILE_STRATEGY = 'imagekit.cachefiles.strategies.Optimistic'
```

If you have specs that *change based on attributes of the source*, that's not going to cut it, though; the file will also need to be generated when those attributes change. Likewise, image generators that don't have sources (i.e. generators that aren't specs) won't cause files to be generated automatically when using the optimistic strategy. (ImageKit can't know when those need to be generated, if not on access.) In both cases, you'll have to trigger the file generation yourself—either by generating the file in code when necessary, or by periodically running the `generateimages` management command. Luckily, ImageKit makes this pretty easy:

```
from imagekit.cachefiles import LazyImageCacheFile

file = LazyImageCacheFile('myapp:profile:avatar_thumbnail', source=source_file)
file.generate()
```

One final situation in which images won't be generated automatically when using the optimistic strategy is when you use a spec with a source that hasn't been registered with it. Unlike the previous two examples, this situation cannot be rectified by running the `generateimages` management command, for the simple reason that the command has no way of knowing it needs to generate a file for that spec from that source. Typically, this situation would arise when using the template tags. Unlike ImageSpecFields, which automatically register all the possible source images with the spec you define, the template tags ("generateimage" and "thumbnail") let you use any spec with any source. Therefore, in order to generate the appropriate files using the `generateimages` management command, you'll need to first register a source group that represents all of the sources you wish to use with the corresponding specs. See *Source Groups* for more information.

5.4 Upgrading from 2.x

ImageKit 3.0 introduces new APIs and tools that augment, improve, and in some cases entirely replace old IK workflows. Below, you'll find some useful guides for migrating your ImageKit 2.0 apps over to the shiny new IK3.

5.4.1 Model Specs

IK3 is chock full of new features and better tools for even the most sophisticated use cases. Despite this, not too much has changed when it comes to the most common of use cases: processing an `ImageField` on a model.

In IK2, you may have used an `ImageSpecField` on a model to process an existing `ImageField`:

```
class Profile(models.Model):
    avatar = models.ImageField(upload_to='avatars')
    avatar_thumbnail = ImageSpecField(image_field='avatar',
                                      processors=[ResizeToFill(100, 50)],
                                      format='JPEG',
                                      options={'quality': 60})
```

In IK3, things look much the same:

```
class Profile(models.Model):
    avatar = models.ImageField(upload_to='avatars')
    avatar_thumbnail = ImageSpecField(source='avatar',
                                      processors=[ResizeToFill(100, 50)],
                                      format='JPEG',
                                      options={'quality': 60})
```

The major difference is that `ImageSpecField` no longer takes an `image_field` kwarg. Instead, you define a `source`.

5.4.2 Image Cache Backends

In IK2, you could gain some control over how your cached images were generated by providing an `image_cache_backend`:

```
class Photo(models.Model):
    ...
    thumbnail = ImageSpecField(..., image_cache_backend=MyImageCacheBackend())
```

This gave you great control over *how* your images are generated and stored, but it could be difficult to control *when* they were generated and stored.

IK3 retains the image cache backend concept (now called cache file backends), but separates the 'when' control out to cache file strategies:

```
class Photo(models.Model):
    ...
    thumbnail = ImageSpecField(...,
                               cachefile_backend=MyCacheFileBackend(),
                               cachefile_strategy=MyCacheFileStrategy())
```

If you are using the IK2 default image cache backend setting:

```
IMAGEKIT_DEFAULT_IMAGE_CACHE_BACKEND = 'path.to.MyImageCacheBackend'
```

IK3 provides analogous settings for cache file backends and strategies:

```
IMAGEKIT_DEFAULT_CACHEFILE_BACKEND = 'path.to.MyCacheFileBackend'
IMAGEKIT_DEFAULT_CACHEFILE_STRATEGY = 'path.to.MyCacheFileStrategy'
```

See the documentation on *cache file backends* and *cache file strategies* for more details.

5.4.3 Conditional model `processors`

In IK2, an `ImageSpecField` could take a `processors` callable instead of an iterable, which allowed processing decisions to made based on other properties of the model. IK3 does away with this feature for consistency's sake (if one kwarg could be callable, why not all?), but provides a much more robust solution: the custom `spec`. See the *advanced usage* documentation for more.

5.4.4 Conditonal `cache_to` file names

IK2 provided a means of specifying custom cache file names for your image specs by passing a `cache_to` callable to an `ImageSpecField`. IK3 does away with this feature, again, for consistency.

There is a way to achieve custom file names by overriding your spec's `cachefile_name`, but it is not recommended, as the spec's default behavior is to hash the combination of `source`, `processors`, `format`, and other spec options to ensure that changes to the spec always result in unique file names. See the documentation on *specs* for more.

5.4.5 Processors have moved to PILKit

Processors have moved to a separate project: PILKit. You should not have to make any changes to an IK2 project to use PILKit–it should be installed with IK3, and importing from `imagekit.processors` will still work.

Index