django-waffle Documentation

Release 0.17.0

James Socol

Jul 11, 2019

Contents

1 Why Waffle? 3
- 1.1 vs Gargoyle . . . 3
- 1.2 Waffle in Production . . . 3

2 Getting Started 5
- 2.1 Requirements . . . 5
- 2.2 Installation . . . 5
- 2.3 Upgrading . . . 7
- 2.4 Configuring Waffle . . . 7

3 Types 9
- 3.1 Flags . . . 9
- 3.2 Switches . . . 12
- 3.3 Samples . . . 13

4 Using Waffle 15
- 4.1 Using Waffle in views . . . 15
- 4.2 Decorating entire views . . . 16
- 4.3 Mixins for Class Based Views . . . 17
- 4.4 Using Waffle in templates . . . 17
- 4.5 Using WaffleJS . . . 19
- 4.6 Managing Waffle data from the command line . . . 20

5 Testing with Waffle 23
- 5.1 Automated testing with Waffle . . . 23
- 5.2 User testing with Waffle . . . 24
- 5.3 Automated testing . . . 26
- 5.4 User testing . . . 26

6 Contributing to Waffle 27
- 6.1 Set Up . . . 27
- 6.2 Writing Patches . . . 27
- 6.3 Submitting Patches . . . 28

7 Waffle's goals 29

8 Roadmap 31

8.1 Present through pre-1.0 . . . 31
8.2 Toward 1.0 . . . 32
8.3 Beyond 1.0 . . . 32

9 Indices and tables **33**

Waffle is feature flipper for Django. You can define the conditions for which a flag should be active, and use it in a number of ways.

Version 0.17.0

Code https://github.com/django-waffle/django-waffle

License BSD; see LICENSE file

Issues https://github.com/django-waffle/django-waffle/issues

Contents:

CHAPTER 1

Why Waffle?

Feature flags are a critical tool for continuously integrating and deploying applications. Waffle is one of several options for managing feature flags in Django applications.

Waffle *aims to*

- provide a simple, intuitive API everywhere in your application;
- cover common use cases with batteries-included;
- be simple to install and manage;
- be fast and robust enough to use in production; and
- minimize dependencies and complexity.

Waffle has an active community and gets fairly steady updates.

1.1 vs Gargoyle

The other major, active feature flag tool for Django is Disqus's Gargoyle. Both support similar features, though Gargoyle offers more options for building custom segments in exchange for some more complexity and requirements.

1.2 Waffle in Production

Despite its pre-1.0 version number, Waffle has been used in production for years at places like Mozilla, Yipit and TodaysMeet.

- Mozilla (Support, MDN, Addons, etc)
- TodaysMeet
- Yipit

(If you're using Waffle in production and don't mind being included here, let me know or add yourself in a pull request!)

CHAPTER 2

Getting Started

2.1 Requirements

Waffle depends only on Django (except for *running Waffle's tests*) but does require certain Django features.

2.1.1 User Models

Waffle requires Django's auth system, in particular it requires both a user model and Django's groups. If you're using a custom user model, this can be accomplished by including Django's PermissionsMixin, e.g.:

```
from django.contrib.auth import models

class MyUser(models.AbstractBaseUser, models.PermissionsMixin):
```

And of `django.contrib.auth` must be in `INSTALLED_APPS`, along with its requirements.

2.1.2 Templates

Waffle provides template tags to check flags directly in templates. Using these requires the `request` object in the template context, which can be easily added with the `request` template context processor:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    # ...
    'django.template.context_processors.request',
    # ...
```

2.2 Installation

After ensuring that the *requirements* are met, installing Waffle is a simple process.

2.2.1 Getting Waffle

Waffle is hosted on PyPI and can be installed with `pip` or `easy_install`:

```
$ pip install django-waffle
$ easy_install django-waffle
```

Waffle is also available on GitHub. In general, `master` should be stable, but use caution depending on unreleased versions.

2.2.2 Settings

Add `waffle` to the `INSTALLED_APPS` setting, and `waffle.middleware.WaffleMiddleware` to `MIDDLEWARE_CLASSES`, e.g.:

```
INSTALLED_APPS = (
    # ...
    'waffle',
    # ...
)

MIDDLEWARE_CLASSES = (
    # ...
    'waffle.middleware.WaffleMiddleware',
    # ...
)
```

Jinja Templates

Changed in version 0.11.

If you're using Jinja2 templates, Waffle provides a Jinja2 extension (`waffle.jinja.WaffleExtension`) to *use Waffle directly from templates*. How you install this depends on which adapter you're using.

With django-jinja, add the extension to the `extensions` list:

```
TEMPLATES = [
    {
        'BACKEND': 'django_jinja.backend.Jinja2',
        'OPTIONS': {
            'extensions': [
                # ...
                'waffle.jinja.WaffleExtension',
            ],
            # ...
        },
        # ...
    },
    # ...
]
```

With jingo, add it to the `JINJA_CONFIG['extensions']` list:

```
JINJA_CONFIG = {
    'extensions': [
```

(continues on next page)

(continued from previous page)

```
        # ...
        'waffle.jinja.WaffleExtension',
    ],
    # ...
}
```

2.2.3 Database Schema

Waffle includes Django migrations for creating the correct database schema. If using Django >= 1.7, simply run the `migrate` management command after adding Waffle to `INSTALLED_APPS`:

```
$ django-admin.py migrate
```

If you're using a version of Django without migrations, you can run `syncdb` to create the Waffle tables.

2.3 Upgrading

2.3.1 From v0.10.x to v0.11

Jinja2 Templates

Waffle no longer supports *jingo's <http://jingo.readthedocs.org/>* automatic helper import, but now ships with a *Jinja2 <http://jinja.pocoo.org/>* extension that supports multiple Jinja2 template loaders for Django. See the *installation docs* for details on how to install this extension.

2.4 Configuring Waffle

There are a few global settings you can define to adjust Waffle's behavior.

WAFFLE_COOKIE The format for the cookies Waffle sets. Must contain `%s`. Defaults to `dwf_%s`.

WAFFLE_FLAG_DEFAULT When a Flag is undefined in the database, Waffle considers it `False`. Set this to `True` to make Waffle consider undefined flags `True`. Defaults to `False`.

WAFFLE_FLAG_MODEL The model that will be use to keep track of flags. Defaults to `waffle.Flag` which allows user- and group-based flags. Can be swapped for a different Flag model that allows flagging based on other things, such as an organization or a company that a user belongs to. Analogous functionality to Django's extendable User models. Needs to be set at the start of a project, as the Django migrations framework does not support changing swappable models after the initial migration.

WAFFLE_SWITCH_DEFAULT When a Switch is undefined in the database, Waffle considers it `False`. Set this to `True` to make Waffle consider undefined switches `True`. Defaults to `False`.

WAFFLE_SAMPLE_DEFAULT When a Sample is undefined in the database, Waffle considers it `False`. Set this to `True` to make Waffle consider undefined samples `True`. Defaults to `False`.

WAFFLE_MAX_AGE How long should Waffle cookies last? (Integer, in seconds.) Defaults to `2529000` (one month).

WAFFLE_READ_FROM_WRITE_DB When calling `*_is_active` methods, Waffle attempts to retrieve a cached version of the object, falling back to the database if necessary. In high- traffic scenarios with multiple databases (e.g. a primary being replicated to a readonly pool) this introduces the risk that a stale version of the object

might be cached if one of these methods is called immediately after an update. Set this to `True` to ensure Waffle always reads Flags, Switches, and Samples from the DB configured for writes on cache misses.

WAFFLE_OVERRIDE Allow *all* Flags to be controlled via the querystring (to allow e.g. Selenium to control their behavior). Defaults to `False`.

WAFFLE_SECURE Whether to set the `secure` flag on cookies. Defaults to `True`.

WAFFLE_CACHE_PREFIX Waffle tries to store objects in cache pretty aggressively. If you ever upgrade and change the shape of the objects (for example upgrading from <0.7.5 to >0.7.5) you'll want to set this to something other than `'waffle:'`. If you're using memcached this should be ASCII only, as that's all it supports.

WAFFLE_CACHE_NAME Which cache to use. Defaults to `'default'`.

WAFFLE_CREATE_MISSING_FLAGS If Waffle encounters a reference to a flag that is not in the database, should Waffle create the flag? If true new flags are created and set to the value of `WAFFLE_FLAG_DEFAULT` Defaults to `False`.

WAFFLE_CREATE_MISSING_SWITCHES If Waffle encounters a reference to a switch that is not in the database, should Waffle create the sample? If true new switchs are created and set to the value of `WAFFLE_SWITCH_DEFAULT` Defaults to `False`.

WAFFLE_CREATE_MISSING_SAMPLES If Waffle encounters a reference to a sample that is not in the database, should Waffle create the sample? If true new samples are created and set to the value of `WAFFLE_SAMPLE_DEFAULT` Defaults to `False`.

WAFFLE_LOG_MISSING_FLAGS If Waffle encounters a reference to a flag that is not in the database, should Waffle log it? The value describes the level of wanted warning, possible values are all levels know by pythons default logging, e.g. `logging.WARNING`. Defaults to `None`.

WAFFLE_LOG_MISSING_SWITCHES If Waffle encounters a reference to a switch that is not in the database, should Waffle log it? The value describes the level of wanted warning, possible values are all levels know by pythons default logging, e.g. `logging.WARNING`. Defaults to `None`.

WAFFLE_LOG_MISSING_SAMPLES If Waffle encounters a reference to a sample that is not in the database,, should Waffle log it? The value describes the level of wanted warning, possible values are all levels know by pythons default logging, e.g. `logging.WARNING`. Defaults to `None`.

CHAPTER 3

Types

Waffle supports three types of feature flippers:

3.1 Flags

Flags are the most robust, flexible method of rolling out a feature with Waffle. Flags can be used to enable a feature for specific users, groups, users meeting certain criteria (such as being authenticated, or superusers) or a certain percentage of visitors.

3.1.1 How Flags Work

Flags compare the current request to their criteria to decide whether they are active. Consider this simple example:

```
if flag_is_active(request, 'foo'):
    pass
```

The *flag_is_active* function takes two arguments, the request, and the name of a flag. Assuming this flag (`foo`) is defined in the database, Waffle will make roughly the following decisions:

- Is `WAFFLE_OVERRIDE` active and if so does this request specify a value for this flag? If so, use that value.
- If not, is the flag set to globally on or off (the *Everyone* setting)? If so, use that value.
- If not, is the flag in *Testing* mode, and does the request specify a value for this flag? If so, use that value and set a testing cookie.
- If not, does the current user meet any of our criteria? If so, the flag is active.
- If not, does the user have an existing cookie set for this flag? If so, use that value.
- If not, randomly assign a value for this user based on the *Percentage* and set a cookie.

3.1.2 Flag Attributes

Flags can be administered through the Django admin site or the *command line*. They have the following attributes:

Name The name of the flag. Will be used to identify the flag everywhere.

Everyone Globally set the Flag, **overriding all other criteria**. Leave as *Unknown* to use other critera.

Testing Can the flag be specified via a querystring parameter? *See below*.

Percent A percentage of users for whom the flag will be active, if no other criteria applies to them.

Superusers Is this flag always active for superusers?

Staff Is this flag always active for staff?

Authenticated Is this flag always active for authenticated users?

Languages Is the `LANGUAGE_CODE` of the request in this list? (Comma-separated values.)

Groups A list of group IDs for which this flag will always be active.

Users A list of user IDs for which this flag will always be active.

Rollout Activate Rollout mode? *See below*.

Note Describe where the flag is used.

A Flag will be active if *any* of the criteria are true for the current user or request (i.e. they are combined with `or`). For example, if a Flag is active for superusers, a specific group, and 12% of visitors, then it will be active if the current user is a superuser *or* if they are in the group *or* if they are in the 12%.

Note: Users are assigned randomly when using Percentages, so in practice the actual proportion of users for whom the Flag is active will probably differ slightly from the Percentage value.

3.1.3 Custom Flag Models

For many cases, the default Flag model provides all the necessary functionality. It allows flagging individual Users and Groups. If you would like flags to be applied to different things, such as companies a User belongs to, you can use a custom flag model.

The functionality uses the same concepts as Django's custom user models, and a lot of this will be immediately recognizable.

An application needs to define a `WAFFLE_FLAG_MODEL` settings. The default is `waffle.Flag` but can be pointed to an arbitrary object.

Note: It is not possible to change the Flag model and generate working migrations. Ideally, the flag model should be defined at the start of a new project. This is a limitation of the *swappable* Django magic. Please use magic responsibly.

The custom Flag model must inherit from *waffle.models.AbstractBaseFlag*. If you want the existing `User` and `Group` based flagging and would like to add more entities to it, you may extend *waffle.models.AbstractUserFlag*.

If you need to reference the class that is being used as the *Flag* model in your project, use the `get_waffle_flag_model()` method. If you reference the Flag a lot, it may be convenient to add `Flag = get_waffle_flag_model()` right below your imports and reference the Flag model as if it had been imported directly.

Example:

```
# settings.py
WAFFLE_FLAG_MODEL = 'myapp.Flag'

# models.py
class Flag(AbstractUserFlag):
    FLAG_COMPANIES_CACHE_KEY = 'FLAG_COMPANIES_CACHE_KEY'
    FLAG_COMPANIES_CACHE_KEY_DEFAULT = 'flag:%s:companies'

    companies = models.ManyToManyField(
        Company,
        blank=True,
        help_text=_('Activate this flag for these companies.'),
    )

    def get_flush_keys(self, flush_keys=None):
        flush_keys = super(Flag, self).get_flush_keys(flush_keys)
        companies_cache_key = get_setting(Flag.FLAG_COMPANIES_CACHE_KEY, Flag.FLAG_
↪COMPANIES_CACHE_KEY_DEFAULT)
        flush_keys.append(keyfmt(companies_cache_key, self.name))
        return flush_keys

    def is_active_for_user(self, user):
        is_active = super(Flag, self).is_active_for_user(user)
        if is_active:
            return is_active

        if getattr(user, 'company_id', None):
            company_ids = self._get_company_ids()
            if user.company_id in company_ids:
                return True

    def _get_company_ids(self):
        cache_key = keyfmt(
            get_setting(Flag.FLAG_COMPANIES_CACHE_KEY, Flag.FLAG_COMPANIES_CACHE_KEY_
↪DEFAULT),
            self.name
        )
        cached = cache.get(cache_key)
        if cached == CACHE_EMPTY:
            return set()
        if cached:
            return cached

        company_ids = set(self.companies.all().values_list('pk', flat=True))
        if not company_ids:
            cache.add(cache_key, CACHE_EMPTY)
            return set()

        cache.add(cache_key, company_ids)
        return company_ids

# admin.py
from waffle.admin import FlagAdmin as WaffleFlagAdmin

class FlagAdmin(WaffleFlagAdmin):
    raw_id_fields = tuple(list(WaffleFlagAdmin.raw_id_fields)  ['companies'])
admin.site.register(Flag, FlagAdmin)
```

3.1.4 Testing Mode

See *User testing with Waffle*.

3.1.5 Rollout Mode

When a Flag is activated by chance, Waffle sets a cookie so the flag will not flip back and forth on subsequent visits. This can present a problem for gradually deploying new features: users can get "stuck" with the Flag turned off, even as the percentage increases.

Rollout mode addresses this by changing the TTL of "off" cookies. When Rollout mode is active, cookies setting the Flag to "off" are session cookies, while those setting the Flag to "on" are still controlled by *WAFFLE_MAX_AGE*.

Effectively, Rollout mode changes the *Percentage* from "percentage of visitors" to "percent chance that the Flag will be activated per visit."

3.1.6 Auto Create Missing

When a flag is evaluated in code that is missing in the database the flag returns the *WAFFLE_FLAG_DEFAULT* value but does not create a flag in the database. If you'd like waffle to create missing flags in the database whenever it encounters a missing flag you can set *WAFFLE_CREATE_MISSING_FLAGS* to `True`. Missing flags will be created in the database and the value of the `Everyone` flag attribute will be set to *WAFFLE_FLAG_DEFAULT* in the auto-created database record.

3.1.7 Log Missing

Wether or not you enabled *Auto Create Missing Flags*, it can be practical to be informed that a flag was or is missing. If you'd like waffle to log a warning, error, ... you can set *WAFFLE_LOG_MISSING_FLAGS* to any level known by Python default logger.

3.2 Switches

Switches are simple booleans: they are on or off, for everyone, all the time. They do not require a request object and can be used in other contexts, such as management commands and tasks.

3.2.1 Switch Attributes

Switches can be administered through the Django admin site or the *command line*. They have the following attributes:

Name The name of the Switch.

Active Is the Switch active or inactive.

Note Describe where the Switch is used.

3.2.2 Auto Create Missing

When a switch is evaluated in code that is missing in the database the switch returns the *WAFFLE_SWITCH_DEFAULT* value but does not create a switch in the database. If you'd like waffle

to create missing switchs in the database whenever it encounters a missing switch you can set *WAFFLE_CREATE_MISSING_SWITCHES* to `True`. Missing switches will be created in the database and the value of the `Active` switch attribute will be set to *WAFFLE_SWITCH_DEFAULT* in the auto-created database record.

3.2.3 Log Missing

Wether or not you enabled *Auto Create Missing Switch*, it can be practical to be informed that a switch was or is missing. If you'd like waffle to log a warning, error, ... you can set *WAFFLE_LOG_MISSING_FLAGS* to any level known by Python default logger.

3.3 Samples

Samples are on a given percentage of the time. They do not require a request object and can be used in other contexts, such as management commands and tasks.

Warning: Sample values are random: if you check a Sample twice, there is no guarantee you will get the same value both times. If you need to rely on the value more than once, you should store it in a variable.

```
# YES
foo_on = sample_is_active('foo')
if foo_on:
    pass

# ...later...
if foo_on:
    pass
```

```
# NO!
if sample_is_active('foo'):
    pass

# ...later...
if sample_is_active('foo'):  # INDEPENDENT of the previous check
    pass
```

3.3.1 Sample Attributes

Samples can be administered through the Django admin site or the *command line*. They have the following attributes:

Name The name of the Sample.

Percent A number from 0.0 to 100.0 that determines how often the Sample will be active.

Note Describe where the Sample is used.

3.3.2 Auto Create Missing

When a sample is evaluated in code that is missing in the database the sample returns the *WAFFLE_SAMPLE_DEFAULT* value but does not create a sample in the database. If you'd like waffle

to create missing samples in the database whenever it encounters a missing sample you can set *WAFFLE_CREATE_MISSING_SAMPLES* to `True`. If *WAFFLE_SAMPLE_DEFAULT* is `True` then the `Percent` attribute of the sample will be created as 100.0 (so that when the sample is checked it always evaluates to `True`). Otherwise the value will be set to 0.0 so that the sample always evaluates to `False`.

3.3.3 Log Missing

Wether or not you enabled *Auto Create Missing Sample*, it can be practical to be informed that a sample was or is missing. If you'd like waffle to log a warning, error, ... you can set *WAFFLE_LOG_MISSING_SAMPLES* to any level known by Python default logger.

CHAPTER 4

Using Waffle

Waffle provides a simple API to check the state of *flags*, *switches*, and *samples* in views and templates, and even on the client in JavaScript.

4.1 Using Waffle in views

Waffle provides simple methods to test *flags*, *switches*, or *samples* in views (or, for switches and samples, anywhere else you're writing Python).

4.1.1 Flags

```
waffle.flag_is_active(request, 'flag_name')
```

Returns `True` if the flag is active for this request, else `False`. For example:

```
import waffle

def my_view(request):
    if waffle.flag_is_active(request, 'flag_name'):
        """Behavior if flag is active."""
    else:
        """Behavior if flag is inactive."""
```

4.1.2 Switches

```
waffle.switch_is_active('switch_name')
```

Returns `True` if the switch is active, else `False`.

4.1.3 Samples

```
waffle.sample_is_active('sample_name')
```

Returns `True` if the sample is active, else `False`.

Warning: See the warning in the *Sample chapter*.

4.2 Decorating entire views

Waffle provides decorators to wrap an entire view in a *flag* or *switch*. (Due to their always-random nature, no decorator is provided for *samples*.)

When the flag or switch is active, the view executes normally. When it is inactive, the view returns a 404. Optionally, you can provide a view or URL name where the decorator can redirect to if you don't want to show a 404 page when the flag or switch is inactive.

4.2.1 Flags

```
from waffle.decorators import waffle_flag

@waffle_flag('flag_name')
def myview(request):
    pass

@waffle_flag('flag_name', 'url_name_to_redirect_to')
def myotherview(request):
    pass
```

4.2.2 Switches

```
from waffle.decorators import waffle_switch

@waffle_switch('switch_name')
def myview(request):
    pass

@waffle_switch('switch_name', 'url_name_to_redirect_to')
def myotherview(request):
    pass
```

4.2.3 Inverting Decorators

Both `waffle_flag` and `waffle_switch` can be reversed (i.e. they will raise a 404 if the flag or switch is *active*, and otherwise execute the view normally) by prepending the name of the flag or switch with an exclamation point: `!`.

```
@waffle_switch('!switch_name')
def myview(request):
    """Only runs if 'switch_name' is OFF."""
```

4.3 Mixins for Class Based Views

Waffle provides mixins to add to Class Based Views.

When the flag or switch is active, or a sample returns True, the view executes normally. When it is inactive, the view returns a 404.

4.3.1 WaffleFlagMixin

```
from waffle.mixins import WaffleFlagMixin

class MyClass(WaffleFlagMixin, View):
    waffle_flag = "my_flag"
```

4.3.2 WaffleSwitchMixin

```
from waffle.mixins import WaffleSwitchMixin

class MyClass(WaffleSwitchMixin, View):
    waffle_switch= "my_switch"
```

4.3.3 WaffleSampleMixin

```
from waffle.mixins import WaffleSampleMixin

class MyClass(WaffleSampleMixin, View):
    waffle_switch= "my_sample"
```

4.4 Using Waffle in templates

Waffle makes it easy to test *flags*, *switches*, and *samples* in templates to flip features on the front-end. It includes support for both Django's built-in templates and for Jinja2.

> **Warning:** Before using samples in templates, see the warning in the *Sample chapter*.

4.4.1 Django Templates

Load the `waffle_tags` template tags:

```
{% load waffle_tags %}
```

In Django templates, Waffle provides three new block types, `flag`, `switch`, and `sample`, that function like `if` blocks. Each block supports an optional `else` to be rendered if the flag, switch, or sample in inactive.

Flags

```
{% flag "flag_name" %}
    flag_name is active!
{% else %}
    flag_name is inactive
{% endflag %}
```

Switches

```
{% switch "switch_name" %}
    switch_name is active!
{% else %}
    switch_name is inactive
{% endswitch %}
```

Samples

```
{% sample "sample_name" %}
    sample_name is active!
{% else %}
    sample_name is inactive
{% endsample %}
```

4.4.2 Jinja Templates

When used with Jinja2, Waffle provides a `waffle` object in the Jinja template context that can be used with normal `if` statements. Because these are normal `if` statements, you can use `else` or `if not` as normal.

Flags

```
{% if waffle.flag('flag_name') %}
    flag_name is active!
{% endif %}
```

Switches

```
{% if waffle.switch('switch_name') %}
    switch_name is active!
{% endif %}
```

Samples

```
{% if waffle.sample('sample_name') %}
    sample_name is active!
{% endif %}
```

4.5 Using WaffleJS

Waffle supports using *flags*, *switches*, and *samples* in JavaScript ("WaffleJS") either via inline script or an external script.

> **Warning:** Unlike samples when used in Python, samples in WaffleJS **are only calculated once** and so are **consistent**.

4.5.1 The WaffleJS `waffle` object

WaffleJS exposes a global `waffle` object that gives access to flags, switches, and samples.

Methods

These methods can be used exactly like their Python equivalents:

- `waffle.flag_is_active(flag_name)`
- `waffle.switch_is_active(switch_name)`
- `waffle.sample_is_active(sample_name)`

Members

WaffleJS also directly exposes dictionaries of each type, where keys are the names and values are `true` or `false`:

- `waffle.FLAGS`
- `waffle.SWITCHES`
- `waffle.SAMPLES`

4.5.2 Installing WaffleJS

As an external script

Using the `wafflejs` view requires adding Waffle to your URL configuration. For example, in your `ROOT_URLCONF`:

```
urlpatterns = patterns('',
    (r'^', include('waffle.urls')),
)
```

This adds a route called `wafflejs`, which you can use with the `url` template tag:

```
<script src="{% url 'wafflejs' %}"></script>
```

As an inline script

To avoid an extra request, you can also use the `wafflejs` template tag to include WaffleJS as an inline script:

```
{% load waffle_tags %}
<script>
  {% wafflejs %}
</script>
```

4.6 Managing Waffle data from the command line

Aside the Django admin interface, you can use the command line tools to manage all your waffle objects.

4.6.1 Flags

Use `manage.py` to change the values of your flags:

```
$ ./manage.py waffle_flag name-of-my-flag --everyone --percent=47
```

Use `--everyone` to turn on and `--deactivate` to turn off the flag. Set a percentage with `--percent` or `-p`. Set the flag on for superusers (`--superusers`), staff (`--staff`) or authenticated (`--authenticated`) users. Set the rollout mode on with `--rollout` or `-r`.

If the flag doesn't exist, add `--create` to create it before setting its values:

```
$ ./manage.py waffle_flag name-of-my-flag --deactivate --create
```

To list all the existing flags, use `-l`:

```
$ ./manage.py waffle_flag -l
Flags:
name-of-my-flag
```

4.6.2 Switches

Use `manage.py` to change the values of your switches:

```
$ ./manage.py waffle_switch name-of-my-switch off
```

You can set a switch to `on` or `off`. If that switch doesn't exist, add `--create` to create it before setting its value:

```
$ ./manage.py waffle_switch name-of-my-switch on --create
```

To list all the existing switches, use `-l`:

```
$ ./manage.py waffle_switch -l
Switches:
name-of-my-switch on
```

4.6.3 Samples

Use `manage.py` to change the values of your samples:

```
$ ./manage.py waffle_sample name-of-my-sample 100
```

You can set a sample to any floating value between `0.0` and `100.0`. If that sample doesn't exist, add `--create` to create it before setting its value:

```
$ ./manage.py waffle_sample name-of-my-sample 50.0 --create
```

To list all the existing samples, use `-l`:

```
$ ./manage.py waffle_sample -l
Samples:
name-of-my-sample: 50%
```

4.6.4 Deleting Data

Use `manage.py` to delete a batch of flags, switches, and/or samples:

```
$ ./manage.py waffle_delete --switches switch_name_0 switch_name_1 --flags flag_name_
↪0 flag_name_1 --samples sample_name_0 sample_name_1
```

Pass a list of switch, flag, or sample names to the command as keyword arguments and they will be deleted from the database.

CHAPTER 5

Testing with Waffle

"Testing" takes on at least two distinct meanings with Waffle:

- Testing your application with automated tools
- Testing your feature with users

For the purposes of this chapter, we'll refer to the former as "automated testing" and the latter as "user testing" for clarity.

5.1 Automated testing with Waffle

Feature flags present a new challenge for writing tests. The test database may not have Flags, Switches, or Samples defined, or they may be non-deterministic.

My philosophy, and one I encourage you to adopt, is that tests should cover *both* code paths, with any feature flags on and off. To do this, you'll need to make the code behave deterministically.

Here, I'll cover some tips and best practices for testing your app while using feature flags. I'll talk specifically about Flags but this can equally apply to Switches or Samples.

5.1.1 Unit tests

Waffle provides three context managers (that can also be used as decorators) in `waffle.testutils` that make testing easier.

- `override_flag`
- `override_sample`
- `override_switch`

All three are used the same way:

```
with override_flag('flag_name', active=True):
    # Only 'flag_name' is affected, other flags behave normally.
    assert waffle.flag_is_active(request, 'flag_name')
```

Or:

```
@override_sample('sample_name', active=True)
def test_with_sample():
    # Only 'sample_name' is affected, and will always be True. Other
    # samples behave normally.
    assert waffle.sample_is_active('sample_name')
```

All three will restore the relevant flag, sample, or switch to its previous state: they will restore the old values and will delete objects that did not exist.

5.1.2 External test suites

Tests that run in a separate process, such as Selenium tests, may not have access to the test database or the ability to mock Waffle values.

For tests that make HTTP requests to the system-under-test (e.g. with Selenium or PhantomJS) the `WAFFLE_OVERRIDE` *setting* makes it possible to control the value of any *Flag* via the querystring.

For example, for a flag named `foo`, we can ensure that it is "on" for a request:

```
GET /testpage?foo=1 HTTP/1.1
```

or that it is "off":

```
GET /testpage?foo=0 HTTP/1.1
```

5.2 User testing with Waffle

Testing a feature (i.e. not *testing the code*) with users usually takes one of two forms: small-scale tests with individuals or known group, and large-scale tests with a subset of production users. Waffle provides tools for the former and has some suggestions for the latter.

5.2.1 Small-scale tests

There are two ways to control a flag for an individual user:

- add their account to the flag's list of users, or
- use testing mode.

Testing mode makes it possible to enable a flag via a querystring parameter (like `WAFFLE_OVERRIDE`) but is unique for two reasons:

- it can be enabled and disabled on a flag-by-flag basis, and
- it only requires the querystring parameter once, then relies on cookies.

If the flag we're testing is called `foo`, then we can enable testing mode, and send users to `oursite.com/testpage?dwft_foo=1` (or `=0`) and the flag will be on (or off) for them for the remainder of their session.

Warning: Currently, the flag **must** be used by the first page they visit, or the cookie will not get set. See #80 on GitHub.

Researchers can send a link with these parameters to anyone and then observe or ask questions. At the end of their session, or when testing mode is deactivated, they will call back to normal behavior.

For a small group, like a company or team, it may be worth creating a Django group and adding or removing the group from the flag.

5.2.2 Large-scale tests

Large scale tests are tests along the lines of "roll this out to 5% of users and observe the relevant metrics." Since "the relevant metrics" is very difficult to define across all sites, here are some thoughts from my experience with these sorts of tests.

Client-side metrics

Google Analytics—and I imagine similar products—has the ability to segment by page or session variables. If you want to A/B test a conversion rate or funnel, or otherwise measure the impact on some client-side metric, using these variables is a solid way to go. For example, in GA, you might do the following to A/B test a landing page:

```
ga('set', 'dimension1', 'Landing Page Version {% flag "new_landing_page" %}2{% else %}
↪1{% endif %}');
```

Similarly you might set session or visitor variables for funnel tests.

The exact steps to both set a variable like this and then to create segments and examine the data will depend on your client-side analytics tool. And, of course, this can be combined with other data and further segmented if you need to.

Server-side metrics

I use StatsD religiously. Sometimes Waffle is useful for load and capacity testing in which case I want to observe timing data or error rates.

Sometimes, it makes sense to create entirely new metrics, and measure them directly, e.g.:

```
if flag_is_active('image-process-service'):
    with statsd.timer('imageservice'):
        try:
            processed = make_call_to_service(data)
        except ServiceError:
            statsd.incr('imageservice.error')
        else:
            statsd.incr('imageservice.success')
else:
    with statsd.timer('process-image'):
        processed = do_inline_processing(data)
```

Other times, existing data—e.g. timers on the whole view—isn't going to move. If you have enough data to be statistically meaningful, you can measure the impact for a given proportion of traffic and derive the time for the new code.

If a flag enabling a refactored codepath is set to 20% of users, and average time has improved by 10%, you can calculate that you've improved the speed by 50%!

You can use the following to figure out the average for requests using the new code. Let t_{old} be the average time with the flag at 0%, t_{total} be the average time with the flag at $p * 100is...$ If you believe my math (you should check it!) then you can measure the average with the flag at 0% to get t_{old} (let's say 1.2 seconds), then at $p * 100$ % (let's say 20%, so $p = 0.2$) to get t_{total} (let's say 1.08 seconds, a 10% improvement) and you have enough to get the average of the new path.

$$t_{new} = 1.2 - \frac{1.2 - 1.08}{0.2} = 0.6$$

Wow, good work!

You can use similar methods to derive the impact on other factors.

5.3 Automated testing

Automated testing encompasses things like unit and integration tests, whether they use the Python/Django unittest framework or an external tool like Selenium.

Waffle is often non-deterministic, i.e. it introduces true randomness to the system-under-test, which is a nightmare for automated testing. Thus, Waffle includes tools to re-introduce determinism in automated test suites.

Read more about automated testing.

5.4 User testing

User testing occurs on both a (relatively) large scale with automated metric collection and on a small, often one-to-one—such as testing sessions with a user and research or turning on a feature within a company or team.

Waffle does what it can to support these kinds of tests while still remaining agnostic about metrics platforms.

Read more about user testing.

CHAPTER 6

Contributing to Waffle

Waffle is pretty simple to hack, and has a decent test suite! Here's how to patch Waffle, add tests, run them, and contribute changes.

Please open a new issue to discuss a new feature before beginning work on it. Not all suggestions are accepted. The *Goals* may help guide which features are likely to be accepted.

6.1 Set Up

Setting up an environment is easy! You'll want `virtualenv` and `pip`, then just create a new virtual environment and install the requirements:

```
$ mkvirtualenv waffle
$ pip install -r requirements.txt
```

Done!

6.2 Writing Patches

Fork Waffle and create a new branch off master for your patch. Run the tests often:

```
$ ./run.sh test
```

Try to keep each branch to a single feature or bugfix.

Note: To update branches, please **rebase** onto master, do not merge master into your branch.

6.3 Submitting Patches

Open a pull request on GitHub!

Before a pull request gets merged, it should be **rebased** onto master and squashed into a minimal set of commits. Each commit should include the necessary code, test, and documentation changes for a single "piece" of functionality.

To be mergable, patches must:

- be rebased onto the latest master,
- be automatically mergeable,
- not break existing tests (TravisCI will run them, too),
- not change existing tests without a *very* good reason,
- add tests for new code (bug fixes should include regression tests, new features should have relevant tests),
- not introduce any new flake8 errors (run `./run.sh lint`),
- document any new features, and
- have a good commit message.

Regressions tests should fail without the rest of the patch and pass with it.

CHAPTER 7

Waffle's goals

Note: This document is a work in progress. See *the roadmap*, too.

Waffle is designed to

- support continuous integration and deployment,
- support feature rollout,
- with minimum set-up time and learning,
- while covering common segments,
- and being fast and robust enough for production use.

Waffle is **not** designed to

- be secure, or be a replacement for permissions,
- cover all potential segments.

CHAPTER 8

Roadmap

Note: This roadmap is subject to change, but represents the rough direction I plan to go. For specific issues, see the current milestones.

Waffle is already a useful library used in many production systems, but it is not done evolving.

8.1 Present through pre-1.0

The immediate future is finishing common segment features and bug fixes.

8.1.1 0.10.2–0.11.x

0.10.2 was primarily a docs overhaul with a major fix to how caching works. It was combined with 0.11. It did include test utilities for consumers.

0.11 updated support, dropping 1.5 and adding 1.8, and overhauled Jinja integration to be compatible with any Jinja2 helper, like jingo or—more future-proof—django-jinja.

0.11.1 is probably the last release of the 0.11.x series. It added support for Django 1.9 without deprecating any other versions.

8.1.2 0.12

0.12 includes a couple of significant refactors designed to pay down some of the debt that's accrued in the past few years.

It also includes support for Django 1.10 and above.

8.1.3 0.13

0.13 drops support for all versions of Django prior to 1.8, including dropping South migrations (and finally being rid of the old issues with them). Along with that, it changes the way settings are configured to be more modern.

0.13 is about closing some long-standing feature gaps, like segmenting by IP and User-Agent.

It also includes finally making a decision about auto-create/data-in-settings.

8.2 Toward 1.0

There are no solid criteria for what makes 1.0 right now, but after 0.13, most outstanding issues will be resolved and Waffle will be in very good shape. There are no plans for a 0.14, so it seems likely that the next step after 0.13 would be some clean-up and finally a 1.0.

8.3 Beyond 1.0

tl;dr: Waffle2 may be a complete break from Waffle.

Waffle is one of the first Python libraries I created, you can see that in the amount of code I left in `__init__.py`. It is also 5 years old, and was created during a different period in my career, and in Django.

There are some philosophical issues with how Waffle is designed. Adding new methods of segmenting users requires at least one new column each, and increasing the cyclomatic complexity. Caching is difficult. The requirements are stringent and no longer realistic (they were created before Django 1.5). The distinction between Flags, Samples, and Switches is confusing and triples the API surface area (Flags can easily act as Switches, less easily as Samples). It is not extensible.

Some challenges also just accrue over time. Dropping support for Django 1.4, the current Extended Support Release, would significantly simplify a few parts.

There is a simplicity to Waffle that I've always appreciated vs, say, Gargoyle. Not least of which is that Waffle works with the built-in admin (or any other admin you care to use). I don't have to write any code to start using Waffle, other than an `if` block. Just add a row and click some checkboxes. Most batteries are included. These are all things that any new version of Waffle must maintain.

Still, if I *want* to write code to do some kind of custom segment that isn't common-enough to belong in Waffle, shouldn't I be able to? (And, if all the core segmenters were built as the same kind of extension, we could lower the bar for inclusion.) If I only care about IP address and percentage, it would be great to skip all the other checks that just happen to be higher in the code.

I have rough sketches of what this looks like, but there are still some significant sticking points, particularly around shoehorning all of this into the existing Django admin. I believe it's *possible*, just potentially *gross*. (Then again, if it's gross underneath but exposes a pleasant UI, that's not ideal, but it's OK.)

The other big sticking point is that this won't be a simple `ALTER TABLE wafle_flag ADD COLUMN` upgrade; things will break.

I've been thinking what Waffle would be like if I designed it from scratch today with slightly different goals, like extensibility. Beyond 1.0, it's difficult to see continuing to add new features without this kind of overhaul.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search