

---

# Gargoyle Documentation

*Release 0.11.0*

**DISQUS**

**May 24, 2017**



---

# Contents

---

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	Enable Gargoyle . . . . .	3
1.2	Nexus Frontend . . . . .	4
1.3	Disabling Auto Creation . . . . .	4
<b>2</b>	<b>Usage</b>	<b>5</b>
2.1	@switch_is_active . . . . .	5
2.2	gargoyle.is_active . . . . .	5
2.3	ifswitch . . . . .	6
2.4	Switch Inheritance . . . . .	6
2.5	Testing Switches . . . . .	7
<b>3</b>	<b>API Reference</b>	<b>9</b>
3.1	Condition Set API reference . . . . .	9
3.2	Manager API reference . . . . .	9
3.3	Signals . . . . .	9
3.4	Test Utilities . . . . .	10



Gargoyle is a platform built on top of Django which allows you to switch functionality of your application on and off based on conditions.

---

**Note:** Gargoyle works with Django 1.2, 1.3, or 1.4.

---



Install using pip:

```
pip install -U gargoyle
```

Or alternatively, easy\_install:

```
easy_install -U gargoyle
```

## Enable Gargoyle

Once you've downloaded the Gargoyle package, you simply need to add it to your `INSTALLED_APPS`:

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

*If you do not use Nexus*, you will also need to enable discovery of `gargoyle.py` modules (which contain `ConditionSets`). The best place to do this is within your `urls.py` file.

```
>>> import gargoyle  
>>> gargoyle.autodiscover()
```

If you do use `gargoyle.py` files and the autodiscovery code, you'll need to ensure your imports are not relative:

```
from __future__ import absolute_import  
  
from gargoyle.conditions import ConditionSet  
# ...
```

## Nexus Frontend

While Gargoyle can be used without a frontend, we highly recommend using [Nexus](#).

Nexus will automatically detect Gargoyle's NexusModule, assuming you're using autodiscovery. If not, you will need to register the module by hand:

```
>>> from gargoyle.nexus_modules import GargoyleModule
>>> nexus.site.register(GargoyleModule, 'gargoyle')
```

(Nexus is a replacement for your Django admin frontend, that works with `django.contrib.admin`)

## Disabling Auto Creation

Under some conditions you may not want Gargoyle to automatically create switches that don't currently exist. To disable this behavior, you may use the `GARGOYLE_AUTO_CREATE` setting in your `settings.py`:

```
GARGOYLE_AUTO_CREATE = False
```

## Default Switch States

The `GARGOYLE_SWITCH_DEFAULTS` setting allows engineers to set the default state of a switch before it's been added via the gargoyle admin interface. In your `settings.py` add something like:

```
GARGOYLE_SWITCH_DEFAULTS = {
    'new_switch': {
        'is_active': True,
        'label': 'New Switch',
        'description': 'When you want the newness',
    },
    'funky_switch': {
        'is_active': False,
        'label': 'Funky Switch',
        'description': 'Controls the funkiness.',
    },
}
```



Gargoyle is designed to work around a very simple API. Generally, you pass in the switch key and a list of instances to check this key against.

### @switch\_is\_active

The simplest way to use Gargoyle is as a decorator. The decorator will automatically integrate with filters registered to the `User` model, as well as IP address (using `RequestConditionSet`):

```
from gargoyle.decorators import switch_is_active

@switch_is_active('my switch name')
def my_view(request):
    return 'foo'
```

In the case of the switch being inactive and you are using the decorator, a 404 error is raised. You may also redirect the user to an absolute URL (relative to domain), or a named URL pattern:

```
# if redirect_to starts with a /, we assume it's a url path
@switch_is_active('my switch name', redirect_to='/my/url/path')

# alternatively use the url mapper
@switch_is_active('my switch name', redirect_to='access_denied')
```

### gargoyle.is\_active

An alternative, more flexible use of Gargoyle is with the `is_active` method. This allows you to perform validation on your own custom objects:

```
from gargoyle import gargoyle

def my_function(request):
    if gargoyle.is_active('my switch name', request):
        return 'foo'
    else:
        return 'bar'

# with custom objects
from gargoyle import gargoyle

def my_method(user):
    if gargoyle.is_active('my switch name', user):
        return 'foo'
    else:
        return 'bar'
```

## ifswitch

If you prefer to use templatetags, Gargoyle provides a helper called `ifswitch` to give you easy conditional blocks based on active switches (for the request):

```
{% load gargoyle_tags %}

{% ifswitch switch_name %}
    switch_name is active!
{% else %}
    switch_name is not active :(
{% endifswitch %}
```

`ifswitch` can also be used with custom objects, like the `gargoyle.is_active` method:

```
{% ifswitch "my switch name" user %}
    "my switch name" is active!
{% endifswitch %}
```

## Switch Inheritance

Switches utilizing the named pattern of `parent:child` will automatically inherit state from their parents. For example, if your switch, `parent:child` is globally enabled, but `parent` is disabled, when `is_active('parent:child')` is called it will return `False`.

A parent switch that has its status set to inherit will return the default value for a switch, which is `False` (the same as disabled).

---

**Note:** Currently inheritance does not combine filters. If your child defines *any* filters, they will override all of the parents.

---

## Testing Switches

Gargoyle includes a context manager, which may optionally be used as a decorator, to give temporary state to a switch on the currently executing thread.

```
from gargoyle.testutils import switches

@switches(my_switch_name=True)
def foo():
    print gargoyle.is_active('my_switch_name')

def foo():
    with switches(my_switch_name=True):
        print gargoyle.is_active('my_switch_name')
```

You may also optionally pass an instance of `SwitchManager` as the first argument:

```
def foo():
    with switches(gargoyle, my_switch_name=True):
        print gargoyle.is_active('my_switch_name')
```



### Condition Set API reference

This document describes the Condition Set API, which allows you to create your own custom switch validation logic.

### Manager API reference

This document describes the Switch Manager API. This is generally referred to as the global `gargoyle` object, which lives in `gargoyle.gargoyle`.

Switches are handled through the `ModelDict` interface, which is registered under the `Switch` model.

### Signals

`gargoyle.signals.switch_added`

This signal is sent when a switch is added (similar to Django's `post_save`, when `created` is `True`).

Example subscriber:

```
def switch_added_callback(sender, request, switch, **extra):
    logging.debug('Switch was added: %r', switch.label)

from gargoyle.signals import switch_added
switch_added.connect(switch_added_callback)
```

`gargoyle.signals.switch_deleted`

This signal is sent when a switch is deleted (similar to Django's `post_delete`).

Example subscriber:

```
def switch_deleted_callback(sender, request, switch, **extra):
    logging.debug('Switch was deleted: %r', switch.label)

from gargoyale.signals import switch_deleted
switch_deleted.connect(switch_deleted_callback)
```

### `gargoyale.signals.switch_updated`

This signal is sent when a switch is updated (similar to Django's `post_save`, when created is False).

Example subscriber:

```
def switch_updated_callback(sender, request, switch, **extra):
    logging.debug('Switch was updated: %r', switch.label)

from gargoyale.signals import switch_updated
switch_updated.connect(switch_updated_callback)
```

### `gargoyale.signals.switch_status_updated`

This signal is sent when a condition is removed from a switch.

Example subscriber:

```
def switch_status_updated_callback(sender, request, switch, status, **extra):
    logging.debug('Switch has updated status: %r; %r', switch.label, status)

from gargoyale.signals import switch_status_updated
switch_status_updated.connect(switch_status_updated_callback)
```

### `gargoyale.signals.switch_condition_added`

This signal is sent when a condition is added to a switch.

Example subscriber:

```
def switch_condition_added_callback(sender, request, switch, condition, **extra):
    logging.debug('Switch has new condition: %r; %r', switch.label, condition)

from gargoyale.signals import switch_condition_added
switch_condition_added.connect(switch_condition_added_callback)
```

### `gargoyale.signals.switch_condition_deleted`

This signal is sent when a condition is removed from a switch.

Example subscriber:

```
def switch_condition_deleted_callback(sender, request, switch, condition,
↪**extra):
    logging.debug('Switch has deleted condition: %r; %r', switch.label, condition)

from gargoyale.signals import switch_condition_deleted
switch_condition_deleted.connect(switch_condition_deleted_callback)
```

## Test Utilities