
Gargoyle Documentation

Release 1.3.0

DISQUS

Aug 04, 2018

Contents

1	Installation	3
1.1	Enable Gargoyle	3
1.2	Nexus Frontend	4
1.3	Disabling Auto Creation	4
2	Usage	5
2.1	@switch_is_active	5
2.2	gargoyle.is_active	5
2.3	Template Tags	6
2.4	Switch Inheritance	6
2.5	Testing Switches	7
2.6	Management Commands	7
3	API Reference	9
3.1	Condition Set API reference	9
3.2	Manager API reference	10
3.3	Signals	11
3.4	Test Utilities	13

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

Install using pip:

```
pip install gargoyle-yplan
```

If you are upgrading from the original to this fork, you will need to run the following first, since the packages clash:

```
pip uninstall django-modeldict gargoyle
```

Failing to do this will mean that `pip uninstall gargoyle` will also erase the files for *gargoyle-yplan*, and similarly for our *django-modeldict* fork.

1.1 Enable Gargoyle

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

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

Gargoyle has autodiscovery similar to Django Admin - it will look in each of your `INSTALLED_APPS` for a `gargoyle` submodule, and import that. You can use this to declare extra `ConditionSets`. If you use such submodules and Python 2.7, you'll need to ensure your imports are not relative in those files:

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

1.2 Nexus Frontend

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

Nexus will automatically detect Gargoyle's `NexusModule`, assuming its autodiscovery is on. If not, you will need to register the module by hand:

```
from gargoyle.nexus_modules import GargoyleModule

nexus.site.register(GargoyleModule, 'gargoyle')
```

1.3 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 your `settings.py`:

```
GARGOYLE_AUTO_CREATE = False
```

1.3.1 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.

2.1 @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 a name that will be passed to reverse()
@switch_is_active('my switch name', redirect_to='access_denied')
```

2.2 gargoyl.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'
```

2.3 Template Tags

If you prefer to use templatetags, Gargoyle provides two helpers called `ifswitch` and `ifnotswitch` 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 %}

{% ifnotswitch other_switch_name %}
    other_switch_name is not active!
{% else %}
    other_switch_name is active!
{% endifnotswitch %}
```

The `else` clauses are optional.

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

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

2.4 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.

2.5 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 test_switches_overrides():
    assert gargoyle.is_active('my_switch_name') # passes

def test_switches_context_manager():
    with switches(my_switch_name=True):
        assert gargoyle.is_active('my_switch_name') # passes
```

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

```
def test_context_manager_alt_gargoyle():
    with switches(gargoyle, my_switch_name=True):
        assert gargoyle.is_active('my_switch_name') # passes
```

2.6 Management Commands

Gargoyle includes management commands which can be useful for task automation (e.g. deployment automation)

2.6.1 add_switch

Adds or updates the specified gargoyle switch.

To create a globally enabled switch or enable an existed switch:

```
manage.py add_switch switch_name
```

To create a disabled switch or disable an existed switch:

```
manage.py add_switch --disabled switch_name
```

2.6.2 remove_switch

Removes the specified gargoyle switch.

Usage:

```
manage.py remove_switch switch_name
```


3.1 Condition Set API reference

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

class `gargoyle.conditions.ConditionSet`

can_execute (*instance*)

Given an instance, returns a boolean of whether this ConditionSet can return a valid condition check.

get_field_value (*instance, field_name*)

Given an instance, and the name of an attribute, returns the value of that attribute on the instance.

Default behavior will map the `percent` attribute to `id`.

get_group_label ()

Returns a string representing a human readable version of this ConditionSet instance.

get_id ()

Returns a string representing a unique identifier for this ConditionSet instance.

get_namespace ()

Returns a string specifying a unique registration namespace for this ConditionSet instance.

has_active_condition (*conditions, instances*)

Given a list of instances, and the conditions active for this switch, returns a boolean representing if any conditional is met, including a non-instance default.

is_active (*instance, conditions*)

Given an instance, and the conditions active for this switch, returns a boolean representing if the feature is active.

3.2 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`.

class `gargoyle.manager.SwitchManager(*args, **kwargs)`

get_all_conditions()

Returns a generator which yields groups of lists of conditions.

```
>>> for set_id, label, field in gargoyle.get_all_conditions():
>>>     print("%(label)s: %(field)s" % (label, field.label))
```

get_condition_set_by_id(*switch_id*)

Given the identifier of a condition set (described in `ConditionSet.get_id()`), returns the registered instance.

get_condition_sets()

Returns a generator yielding all currently registered `ConditionSet` instances.

is_active(*key*, **instances*, *kwargs*)**

Returns True if any of instances match an active switch. Otherwise returns False.

```
>>> gargoyle.is_active('my_feature', request)
```

register(*condition_set*)

Registers a condition set with the manager.

```
>>> condition_set = MyConditionSet()
>>> gargoyle.register(condition_set)
```

unregister(*condition_set*)

Unregisters a condition set with the manager.

```
>>> gargoyle.unregister(condition_set)
```

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

class `gargoyle.models.Switch(*args, **kwargs)`

Stores information on all switches. Generally handled through an instance of `ModelDict`, which is registered under the global `gargoyle` namespace.

value is stored with by type label, and then by column:

```
>>> {
>>>   namespace: {
>>>     id: [[INCLUDE, 0, 50], [INCLUDE, 'string']] // 50% of users
>>>   }
>>> }
```

exception `DoesNotExist`

exception `MultipleObjectsReturned`

add_condition(*manager*, *condition_set*, *field_name*, *condition*, *exclude=False*, *commit=True*)

Adds a new condition and registers it in the global `gargoyle` switch manager.

If `commit` is `False`, the data will not be written to the database.

```
>>> switch = gargoyles['my_switch']
>>> condition_set_id = condition_set.get_id()
>>> switch.add_condition(condition_set_id, 'percent', '0-50', exclude=False)
```

clear_conditions (*manager, condition_set, field_name=None, commit=True*)

Clears conditions given a set of parameters.

If `commit` is `False`, the data will not be written to the database.

Clear all conditions given a `ConditionSet`, and a field name:

```
>>> switch = gargoyles['my_switch']
>>> condition_set_id = condition_set.get_id()
>>> switch.clear_conditions(condition_set_id, 'percent')
```

You can also clear all conditions given a `ConditionSet`:

```
>>> switch = gargoyles['my_switch']
>>> condition_set_id = condition_set.get_id()
>>> switch.clear_conditions(condition_set_id)
```

get_active_conditions (*manager*)

Returns a generator which yields groups of lists of conditions.

```
>>> for label, set_id, field, value, exclude in gargoyles.get_all_conditions():
>>>     print("%(label)s: %(field)s = %(value)s (exclude: %(exclude)s)" %
↳(label, field.label, value, exclude))
```

remove_condition (*manager, condition_set, field_name, condition, commit=True*)

Removes a condition and updates the global `gargoyles` switch manager.

If `commit` is `False`, the data will not be written to the database.

```
>>> switch = gargoyles['my_switch']
>>> condition_set_id = condition_set.get_id()
>>> switch.remove_condition(condition_set_id, 'percent', [0, 50])
```

3.3 Signals

`gargoyles.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 gargoyles.signals import switch_added
switch_added.connect(switch_added_callback)
```

`gargoyles.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 updated in 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)
```


3.4 Test Utilities

```
class gargoyl.testutils.SwitchContextManager (gargoyle=<SimpleLazyObject:
                                         <SwitchManager:
                                         'gargoyle.models.Switch'>
                                         (dict_values([]))>>, **keys) <class
```

Allows temporarily enabling or disabling a switch.

Ideal for testing.

```
>>> @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'))
```

Can also wrap unittest classes, which includes Django's TestCase classes:

```
>>> @switches(my_switch_name=True)
... class MyTests(TestCase):
...     @classmethod
...     def setUpTestData(cls):
...         # my_switch_name is True here
...
...     def test_foo(self):
...         # ... and here
```


A

add_condition() (gargoyle.models.Switch method), 10

C

can_execute() (gargoyle.conditions.ConditionSet method), 9

clear_conditions() (gargoyle.models.Switch method), 11

ConditionSet (class in gargoyle.conditions), 9

G

get_active_conditions() (gargoyle.models.Switch method), 11

get_all_conditions() (gargoyle.manager.SwitchManager method), 10

get_condition_set_by_id() (gargoyle.manager.SwitchManager method), 10

get_condition_sets() (gargoyle.manager.SwitchManager method), 10

get_field_value() (gargoyle.conditions.ConditionSet method), 9

get_group_label() (gargoyle.conditions.ConditionSet method), 9

get_id() (gargoyle.conditions.ConditionSet method), 9

get_namespace() (gargoyle.conditions.ConditionSet method), 9

H

has_active_condition() (gargoyle.conditions.ConditionSet method), 9

I

is_active() (gargoyle.conditions.ConditionSet method), 9

is_active() (gargoyle.manager.SwitchManager method), 10

R

register() (gargoyle.manager.SwitchManager method), 10

remove_condition() (gargoyle.models.Switch method), 11

S

Switch (class in gargoyle.models), 10

Switch.DoesNotExist, 10

Switch.MultipleObjectsReturned, 10

SwitchContextManager (class in gargoyle.testutils), 13

SwitchManager (class in gargoyle.manager), 10

U

unregister() (gargoyle.manager.SwitchManager method), 10