django-conditions Documentation

Release 0.9.17

Lucas Connors

Feb 11, 2022

Contents

1 About **1**

2 Installation **3**

3 Contents **5**

3.1 Basic Usage 5
3.2 Condition Lists 6
3.3 Operators and Operands 7
3.4 Using Keys 8
3.5 Providing Additional Context 8
3.6 Handling Exceptions 9

CHAPTER 1

About

django-conditions is a Django app that allows creation of conditional logic in admin. Some possible use cases:

- Segment your user base into cohorts with targeted messaging
- Provide different rewards to users depending on their expected value
- In a game, define the winning objectives of a mission/quest
- and many more. . .

django-conditions supports Django 2.2-3.2 and Python 3.7-3.10.

CHAPTER 2

Installation

First install the `django-conditions` package:

```
pip install django-conditions
```

Then add `conditions` to your `INSTALLED_APPS` setting:

```
## settings.py
INSTALLED_APPS = [
    ...
    'conditions',
]
```

CHAPTER 3

Contents

3.1 Basic Usage

Start by defining a condition in code:

```
## condition_types.py
from conditions import Condition

class FullName(Condition):
    # The name that appears in the db and represents your condition
    condstr = 'FULL_NAME'

    # Normal conditions define eval_bool, which takes in a user
    # and returns a boolean
    def eval_bool(self, user, **kwargs):
        return bool(user.first_name and user.last_name)
```

Then add a ConditionsField to your model:

```
## models.py
from django.db import models
from conditions import ConditionsField, conditions_from_module
import condition_types

class Campaign(models.Model):
    text = models.TextField()

    # The ConditionsField requires the definitions of all possible conditions
    # conditions_from_module can take an imported module and sort this out for you
    target = ConditionsField(definitions=conditions_from_module(condition_types))
```

In the model's change form on admin, you can enter JSON to represent when you want your condition to be satisfied.

```
{
    "all": ["FULL_NAME"]
}
```

Now you can use the logic you created in admin to determine the outcome of an event:

```
## views.py
from django.http import HttpResponse
from conditions import eval_conditions
from models import Campaign

def profile(request):
    for campaign in Campaign.objects.all():
        if eval_conditions(campaign, 'target', request.user):
            return HttpReponse(campaign.text)

    return HttpResponse("Nothing new to see.")
```

Use django-conditions in your Django projects to change simple logic without having to re-deploy, and pass on the power to product managers and other non-engineers.

3.2 Condition Lists

Once you've created a few Conditions, you can combine them in many different ways.

To require multiple conditions to be satisfied at the same time, use an "all" list:

```
{
    "all": ["FULL_NAME", "ACTIVE", "SUPERUSER"]
}
```

To allow just one of a set of conditions, use an "any" list:

```
{
    "any": ["FULL_NAME", "FB_CONNECTED", "EMAIL_VERIFIED"]
}
```

Of course the lists may be nested:

```
{
    "all": [
        "ACTIVE",
        "SUPERUSER",
        {
            "any": [
                "FULL_NAME",
                "FB_CONNECTED",
                "EMAIL_VERIFIED"
            ]
        }
    ]
}
```

You may also add `NOT` to the beginning of any condition to evaluate its negation:

```
{
    "all": ["FULL_NAME", "ACTIVE", "NOT SUPERUSER", "NOT STAFF"]
}
```

3.3 Operators and Operands

Conditions that subclass CompareCondition are slightly more advanced than regular Conditions. You can use them to make comparisons with numbers (or other operands).

```
from conditions import CompareCondition

class Level(CompareCondition):
    condstr = 'LEVEL'

    # CompareConditions define eval_operand instead of eval_bool
    # which returns an operand instead
    def eval_operand(self, user, **kwargs):
        return user.profile.level
```

In JSON, numbers can be compared using the normal boolean operators you see in Python (<, <=, ==, !=, >, and >=):

```
{
    "all": ["FULL_NAME", "LEVEL >= 5"]
}
```

By default, a CompareCondition expects a float as the operand, but that can be overwritten by defining the cast_operand attribute. You can also define your own operators by writing your own operators function:

```
import datetime
from conditions import CompareCondition

class DateJoined(CompareCondition):
    condstr = 'DATE_JOINED'
    cast_operand = lambda self, timestamp: datetime.datetime.strptime(timestamp, "%m/
↪%d/%Y")

    @classmethod
    def operators(cls):
        return {
            '<': datetime.datetime.__lt__,
            '<=': datetime.datetime.__le__,
            '==': datetime.datetime.__eq__,
            '!=': datetime.datetime.__ne__,
            '>=': datetime.datetime.__ge__,
            '>': datetime.datetime.__gt__,
        }

    def eval_operand(self, user, **kwargs):
        return user.date_joined.strftime("%m/%d/%Y")
```

In the admin interface, an appropriate example is randomly generated for each Condition available. When the operand is a number, a number will be generated randomly for the example automatically. However, with other types of operands this isn't possible so the example will simply show `SOME_OPERAND_HERE`. If you like, you can instead show an appropriate example randomly selected from a list you define:

```
class DateJoined(CompareCondition):
    condstr = 'DATE_JOINED'
    cast_operand = timestamp2datetime
    operand_examples = ['04/23/1995', '01/01/2015', '08/13/2014',]

    ...
```

3.4 Using Keys

Sometimes you want to generalize a Condition so that the user entering the Conditions JSON can provide an arbitrary string that changes how the Condition gets evaluated slightly. In these cases, you can use a key:

```
class EmailDomain(Condition):
    condstr = 'EMAIL_DOMAIN'

    def eval_bool(self, user, **kwargs):
        domain = user.email.split('@')[1]
        return domain == self.key
```

```
{
    "any": ["EMAIL_DOMAIN gmail.com", "EMAIL_DOMAIN yahoo.com"]
}
```

When the Conditions `"EMAIL_DOMAIN gmail.com"` and `"EMAIL_DOMAIN yahoo.com"` get evaluated, `self.key` will contain the strings “gmail.com” and “yahoo.com” respectively.

Of course, the admin interface once again does not know what keys are appropriate so by default, the random example will simply say `SOME_KEY_HERE`. You can define `key_examples` as a list similar to `operand_examples`, or if the set of all possible keys is finite, you may define `keys_allowed` to actually restrict the user entering the Conditions JSON to one of the options from a list:

```
class EmailDomain(Condition):
    condstr = 'EMAIL_DOMAIN'
    key_examples = ['gmail.com', 'yahoo.com', 'hotmail.com']

    ...
```

3.5 Providing Additional Context

Sometimes you want to evaluate a condition on more information than just a user object. For these cases, you can provide any arbitrary keyword arguments to `eval_conditions`:

```
## views.py
from django.http import HttpResponse
from conditions import eval_conditions
from models import Campaign

def room(request, room_num):
    for campaign in Campaign.objects.all():
        if eval_conditions(campaign, 'target', request.user, room=room_num):
            return HttpReponse(campaign.text)
```

(continues on next page)

(continued from previous page)

```
    return HttpResponse("Nothing in this room.")
```

Any keyword arguments are then passed through to every condition that needs to be evaluated:

```
class InRoom(Condition):
    condstr = 'IN_ROOM'

    def eval_bool(self, user, **kwargs):
        room_num = kwargs['room']
        return room_num == int(self.key)
```

3.6 Handling Exceptions

When an exception is raised in the evaluation of a Condition, it fails silently by returning False.

```
# An example where the condition will raise a ValueError

class Broken(Condition):
    condstr = 'BROKEN'

    def eval_bool(self, user, **kwargs):
        raise ValueError("This condition is broken.")
```

Failing silently often avoids major problems, but generally, you still want to know when these issues are popping up. django-conditions provides a simple way using Python's built-in logging framework.

To record these exceptions, add a logger for `condition` in the `LOGGING` configuration dictionary, and handle it however you like. Here is an example, modified from Django's logging page:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': '/path/to/django/debug.log',
        },
    },
    'loggers': {
        'django.request': {
            'handlers': ['file'],
            'level': 'DEBUG',
            'propagate': True,
        },
        'condition': {
            'handlers': ['file'],
            'level': 'DEBUG',
        }
    },
}
```