django-clausula Documentation

Release 1.0.0-rc1

Dominik Kozaczko

August 30, 2012

CONTENTS

The purpose of this app is to allow adding dynamic conditions to relations between objects.

CHAPTER

ONE

NAME

“Clausula” means “condition” in Latin.

CHAPTER

TWO

HOW IS THIS USEFUL?

Let’s say you develop an application that will present content to user based on various conditions. Gamification comes to mind. Let’s say you want to set the conditions on a per-user or per-group basis. For example girls get happy hours on Saturday and boys get them on Thursdays. Hardcoding that would be a bit tricky if you tend to change your mind frequently.

CHAPTER

THREE

USAGE

After installing the package and adding 'clausula' to INSTALLED_APPS you should define some abstract conditions you'd like to use.

Do this by creating a file `conditions.py` in your app. Then you need to import `clauses` registry, define your conditions and register them.

Each condition is simply a function that fullows these rules:

- it takes one mandatory argument: an `object` (a `Condition` instance)
- it silently accepts any number of optional arguments (*args, **kwargs)
- it returns a boolean value

The `object` is guaranteed to have a `param` attribute which holds a string which can be used to compute returned boolean value. There can also be some relations available as `Condition` subclasses `django.db.models.Model`. Feel free to experiment and find some hackish uses for this package.

CHAPTER

FOUR

FULL EXAMPLE

You run a virutal pub and want to have lower prices on one day.

`conditions.py`:

```python
from clausula import clauses

def day_of_week_clause(obj, *args, **kwargs):
    import datetime
    weekday = datetime.date.today().weekday()
    if weekday == int(obj.param):
        return True
    return False

clauses.register(day_of_week_clause, "checks day of week")
```

Now let's see our `models.py`:

```python
from django.db import models
from clausula.models import Condition

class Beverage(models.Model):
    name = models.CharField(max_length=30)
    normal_price = models.DecimalField(max_digits=7, decimal_places=2)

    def __unicode__(self):
        return self.name


class Redeem(models.Model):
    value = models.DecimalField(max_digits=7, decimal_places=2)
    beverage = models.ForeignKey(Beverage)
    condition = models.ForeignKey(Condition)

    def __unicode__(self):
        return "%s %s %s" % (self.value, self.beverage, self.condition)
```

A bit of sugar in `admin.py`:

```python
from django.contrib import admin
from .models import (Beverage, Redeem)

class RedeemInline(admin.TabularInline):
    model = Redeem


```

```python
class BeverageAdmin(admin.ModelAdmin):
    inlines = [RedeemInline]


admin.site.register(Beverage, BeverageAdmin)
```

Then you should run `./manage.py syncdb && ./manage.py runserver`, go to the admin page and add a Condition. You'll see "checks day of week" in `Clause` list. Fill the name and give a day number.

You should also add a `Beverage` with redeem triggered by your `On Sunday` condition.

This is an example template to see if it works:

```
{% load clausula_tags %}
{% for brew in beverages %}
    {{ brew.name }}:
    {% if brew.redeem_set.all %}
        {% for redeem in brew.redeem_set.all %}
            {% check redeem.condition as result %}
            {% if result %}
                (condition met)
            {% else %}
                (condition not met)
            {% endif %}
        {% endfor %}
    {% else %}
        (no redeems)
    {% endif %}
{% endfor %}
```

All that's left is `urls.py` to tie it all together - let it be an exercise for you ;) Anyway you can always just fire `./manage.py runserver` from the `example_project` directory and browse to `http://127.0.0.1:8000/brew/` :)

Now play with `param` and check if it works properly. Add some more objects. Try writing another function and swap it with the one you used in example. Does it trigger properly? Experiment.

CHAPTER

FIVE

FEEDBACK

If you have any ideas how to extend functionality of this little package, fork it on github and make a pull request or simply file a feature request.

CHAPTER

SIX

VERSIONING

I'd like to follow Semantic Version guidelines.

Releases will be numbered with the following format:

`<major>.<minor>.<patch>`

And constructed with the following guidelines:

- Breaking backward compatibility bumps the major (and resets the minor and patch)
- New additions without breaking backward compatibility bumps the minor (and resets the patch)
- Bug fixes and misc changes bumps the patch
- Major version `0` means early development stage

For more information on SemVer, please visit http://semver.org/.

CHAPTER

SEVEN

AUTODOC

7.1 *django-clausula* internal documentation

django-clausula is a pluggable app to allow adding various conditions via Django's admin site. Abstracts for these conditions are defined in source code and registered in similar fashion as admin models.

class `clausula.base.`**`ConditionCache`**

This is a typical registering class.

`choices``()`

This method is used to populate choices field in the admin.

Return type nested list in CHOICES format

`register``(`*condition*, *name=None*`)`

This method is used to register your statement to the global *clauses* list registry.

Parameters

- **condition** – a function that will be called to resolve the condition
- **name** (*string*) – a name that will be shown in the admin

`clausula.base.`**`autodiscover`**`()`

This is shamelessly taken from django.admin.

This method is used to populate the global *clauses* object with whatever conditions developers put in their apps.

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX