Django Computed Property Documentation

Release 0.1

Jason Brechin

Jul 28, 2019

Contents

Computed Property fields for Django models, inspired by Google Cloud NDB

CHAPTER 1

Prerequisites

`django-computed-property` supports (i.e. is tested on) Django 1.8 - 2.2 and trunk on Python 2.7, 3.4, 3.5, 3.6, 3.7, pypy, and pypy3.

SQLite and Postgres are currently tested, but any Django database backend should work.

CHAPTER 2

Installation

`django-computed-property` is available on PyPI. Install it with:

```
pip install django-computed-property
```

CHAPTER 3

Usage

Add computed_property to your list of *INSTALLED_APPS* in *settings.py*:

```
INSTALLED_APPS = [
    ...
    'computed_property'
]
```

Then, simply import and use the included field classes in your models:

```
from django.db import models
from computed_property import ComputedTextField


class MyModel(models.Model):
    name = ComputedTextField(compute_from='calculation')

    @property
    def calculation(self):
        return 'some complicated stuff'
```

You can read values from the `name` field as usual, but you may not set the field's value. When the field is accessed and when a model instance is saved, it will compute the field's value using the provided callable (function/lambda), property name, or attribute name.

compute_from can be a reference to a function that takes a single argument (an instance of the model), or a string referring to a field, property, or other attribute on the instance.

Note: It is important to note that your computed field data will not immediately be written to the database. You must (re-)save all instances of your data to have the computed fields populated in the database. Until you do so, you will be able to access those fields when you load an instance of the model, but you will not benefit from their queryability.

One way you could do this is in a data migration, using something like:

```
for instance in MyModel.objects.all().iterator():
    instance.save()
```

3.1 Field types

Several other field classes are included: `ComputedCharField`, `ComputedEmailField`, `ComputedIntegerField`, `ComputedDateField`, `ComputedDateTimeField`, and others. All field classes accept the same arguments as their non-Computed versions.

To create an Computed version of some other field class, inherit from both `ComputedField` and the other field class:

```
from computed_property import ComputedField
from somewhere import MyField

class MyComputedField(ComputedField, MyField):
    pass
```

CHAPTER 4

Contributing

Please also refer to the contributing docs in the repository.

On top of the above, developers please consider the following:

4.1 Reporting Issues or Suggestions

If you see an issue with the library or have a request for new functionality, please file an issue in GitHub.

Please include specifics about which version of Python you're using, as well as which version of the django-computed-property library you're using.

Whenever possible, include a code sample that demonstrates the issue you're seeing or the desired developer experience.

4.2 Adding Test Cases

The complete test suite is run using `tox`. This is how tests are run on Travis-CI, it includes all supported Python versions, all supported databases back ends, and all supported Django versions. Arguably not what you would want to do, each time you add a test case, or make a minor change.

To run the test suite in just one version of Python, against sqlite3, and using one chosen Django version, you still use `tox`, instructing it to just test that single configuration combination.

You can enumerate all the build configurations with `tox -l`. From that list, you can choose the combination of python-django-database (or the `py37-docs` or `py37-flake8` builds) to run.

For example:

```
$ tox -e py37-django111-sqlite
```

Tox generally just requires that the version of python you're testing against is installed on your system. It will take care of creating the test environment from the configuration information in `tox.ini`.