By **Vitor Freitas**

I'm a passionate software developer and researcher from Brazil, currently living in Finland. I write about Python, Django and Web Development on a weekly basis. Read more.

TIPS

Django Tips #22 Designing Better Models

Feb 10, 2018 · 8 minutes read · 14 comments · 4,986 views

In this post, I will share some tips to help you improve the design of your Django Models. Many of those tips are related to naming conventions, which can improve a lot the readability of your code.

The PEP8 is widely used in the Python ecosystem (Django included). So it's a good idea to use it in your own projects.

Besides PEP8, I like to follow Django's Coding Style which is a guideline for people writing code for inclusion in the Django code base itself.

Below, an overview of the items we are going to explore:

Naming Your Models

The model definition is a class, so always use **CapWords** convention (no underscores). E.g. `User`, `Permission`, `ContentType`, etc.

For the model's attributes use **snake_case**. E.g. `first_name`, `last_name`, etc.

Example:

```
from django.db import models

class Company(models.Model):
    name = models.CharField(max_length=30)
    vat_identification_number = models.CharField(max_length=20)
```

Always name your models using **singular**. Call it `Company` instead of `Companies`. A model definition is the representation of a single object (the object in this example is a **company**), and not a collection of companies.

This usually cause confusion because we tend to think in terms of the database tables. A model will eventually be translated into a table. The table is correct to be named using its plural form because the table represents a collection of objects.

In a Django model, we can access this collection via `Company.objects`. We can renamed the `objects` attribute by defining a `models.Manager` attribute:

```
from django.db import models

class Company(models.Model):
    # ...
    companies = models.Manager()
```

So with that we would access the collection of companies as `Company.companies.filter(name='Google')`. But I usually don't go there. I prefer keeping the `objects` attribute there for consistency.

Model Style Ordering

The Django Coding Style suggests the following order of inner classes, methods and attributes:

- If **choices** is defined for a given model field, define each choice as a tuple of tuples, with an all-uppercase name as a class attribute on the model.
- All database fields
- Custom manager attributes
- `class Meta`
- `def __str__()`
- `def save()`
- `def get_absolute_url()`
- Any custom methods

Example:

```python
from django.db import models
from django.urls import reverse

class Company(models.Model):
    # CHOICES
    PUBLIC_LIMITED_COMPANY = 'PLC'
    PRIVATE_COMPANY_LIMITED = 'LTD'
    LIMITED_LIABILITY_PARTNERSHIP = 'LLP'
    COMPANY_TYPE_CHOICES = (
        (PUBLIC_LIMITED_COMPANY, 'Public limited company'),
        (PRIVATE_COMPANY_LIMITED, 'Private company limited by shares'),
        (LIMITED_LIABILITY_PARTNERSHIP, 'Limited liability partnership'),
    )

    # DATABASE FIELDS
    name = models.CharField('name', max_length=30)
    vat_identification_number = models.CharField('VAT', max_length=20)
    company_type = models.CharField('type', max_length=3, choices=COMPANY_TYPE

    # MANAGERS
    objects = models.Manager()
    limited_companies = LimitedCompanyManager()

    # META CLASS
    class Meta:
        verbose_name = 'company'
        verbose_name_plural = 'companies'

    # TO STRING METHOD
    def __str__(self):
        return self.name

    # SAVE METHOD
    def save(self, *args, **kwargs):
        do_something()
        super().save(*args, **kwargs)  # Call the "real" save() method.
        do_something_else()

    # ABSOLUTE URL METHOD
    def get_absolute_url(self):
        return reverse('company_details', kwargs={'pk': self.id})

    # OTHER METHODS
    def process_invoices(self):
        do_something()
```

Reverse Relationships

related_name

The `related_name` attribute in the `ForeignKey` fields is extremely useful. It let's us define a meaningful name for the reverse relationship.

Rule of thumb: **if you are not sure what would be the `related_name`, use the plural of the model holding the `ForeignKey`.**

```
class Company:
    name = models.CharField(max_length=30)

class Employee:
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    company = models.ForeignKey(Company, on_delete=models.CASCADE, related_nai
```

That means the `Company` model will have a special attribute named `employees`, which will return a `QuerySet` with all employees instances related to the company.

```
google = Company.objects.get(name='Google')
google.employees.all()
```

You can also use the reverse relationship to modify the `company` field on the `Employee` instances:

```
vitor = Employee.objects.get(first_name='Vitor')
google = Company.objects.get(name='Google')
google.employees.add(vitor)
```

related_query_name

This kind of relationship also applies to query filters. For example, if I wanted to list all companies that employs people named 'Vitor', I could do the following:

```
companies = Company.objects.filter(employee__first_name='Vitor')
```

If you want to customize the name of this relationship, here is how we do it:

```
class Employee:
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    company = models.ForeignKey(
        Company,
        on_delete=models.CASCADE,
        related_name='employees',
        related_query_name='person'
    )
```

Then the usage would be:

```
companies = Company.objects.filter(person__first_name='Vitor')
```

To use it consistently, `related_name` goes as **plural** and `related_query_name` goes as **singular**.

Blank and Null Fields

I've written about the differences between Blank and Null fields in another post, but I will try to summarize it here:

- **Null**: It is database-related. Defines if a given database column will accept null values or not.
- **Blank**: It is validation-related. It will be used during forms validation, when calling `form.is_valid()`.

Do not use `null=True` for text-based fields that are optional. Otherwise, you will end up having two possible values for "no data," that is: **None** and an **empty string**. Having two possible values for "no data" is redundant. The Django convention is to use the empty string, not NULL.

Example:

```
# The default values of `null` and `blank` are `False`.
class Person(models.Model):
    name = models.CharField(max_length=255)  # Mandatory
    bio = models.TextField(max_length=500, blank=True)  # Optional (don't put
    birth_date = models.DateField(null=True, blank=True) # Optional (here you
```

Further Reading

Models definition is one of the most important parts of your application. Something that makes all the difference is defining the field types properly. Make sure to review the Django models field types to know your options. You can also define custom field types.

If you are interested in code conventions, I suggest having a look on Django's Coding Style. I've also published an tutorial about the flake8 library which helps you check for PEP8 issues in your code.

That's it for today! You can also subscribe to my newsletter to receive updates from the blog.

Related Posts