Лучшие практики.
Работа с моделями в Django .

ITcoder

Итак, приступим:

1. Правильное определение названия модели

Обычно рекомендуется использовать существительные единственного числа для именования моделей, например: **User**, **Post**, **Article.** То есть, последний компонент названия должен быть существительным, например: **Some New Shiny Item.** Правильно использовать единственное число, когда одна единица модели не содержит информации о нескольких объектах.

2. Название полей связей(связанных объектов)

Для таких связей, как **ForeignKey**, **OneToOneKey**, **ManyToMany**, иногда лучше указать имя. Представьте, что есть модель **Article**, **-** в которой одним из связей является **ForeignKey** для модели **User.** Если это поле содержит информацию об авторе статьи, то **author** будет более подходящим именем, чем **user.**

3. Правильное указание названия связанного объекта.

Целесообразно указывать связанное имя во множественном числе, так как обращение к связанному имени возвращает **queryset.** Пожалуйста, задавайте адекватные связанные имена. В большинстве случаев имя модели во множественном числе будет правильным.

Например:

```
class Owner(models.Model):
        pass
class Item(models.Model):
    owner = models.ForeignKey(Owner, related_name='items')
```

4. Не используйте ForeignKey с unique=True

Нет смысла использовать **ForeignKey** с **unique=True,** поскольку для таких случаев существует **OneToOneField.**

5. Порядок атрибутов и методов в модели

Предпочтительный порядок атрибутов и методов в модели.

- константы (для выбора и другие)
- поля модели
- указание пользовательского менеджера
- **meta**
- **def __unicode__ (python 2)** или **def __str__ (python 3)**
- другие специальные методы
- **def clean**
- **def save**
- **def get_absolut_url**
- другие методы

Обратите внимание, что приведенный порядок был взят из документации и немного расширен.

6. Добавление модели через миграцию

Если вам нужно добавить модель, то, создав класс модели, выполните последовательно команды **manage.py makemigrations** и **migrate** (или используйте **South** для **Django 1.6** и ниже).

7. Денормализация

Не следует допускать бездумного использования денормализации в реляционных базах данных. Всегда старайтесь избегать этого, за исключением тех случаев, когда вы денормализуете данные сознательно по какой-либо причине (например, для повышения производительности). Если на этапе проектирования базы данных вы понимаете, что вам необходимо денормализовать большую часть данных, хорошим вариантом может стать использование **NoSQL**. Однако, если большая часть данных не требует денормализации, чего нельзя избежать, подумайте о реляционной базе с **JsonField** для хранения некоторых данных.

8. BooleanField

Не используйте **null=True** или **blank=True** для **BooleanField**. Следует также отметить, что для таких полей лучше указывать значения по умолчанию. Если вы понимаете, что поле может оставаться пустым, вам нужно **NullBooleanField**.

9. Бизнес-логика в моделях

Лучшее место для размещения бизнес-логики вашего проекта - это модели, а именно модели методов и менеджер моделей. Возможно, что модели методов могут вызывать только некоторые методы/функции. Если неудобно или невозможно разместить логику в моделях, необходимо заменить ее формы или сериализаторы в выполнении задач.

10. Дублирование полей в ModelForm

Не дублируйте поля модели в **ModelForm** или **ModelSerializer** без необходимости. Если вы хотите указать, что форма использует все поля модели, используйте **MetaFields.** Если вам нужно переопределить виджет для поля, при этом в этом поле больше ничего не нужно менять, используйте **Meta widgets** для указания виджетов.

11. Не используйте ObjectDoesNotExist

При использовании **ModelName.DoesNotExist** вместо **ObjectDoesNotExist** перехват исключений становится более специализированным, что является хорошей практикой.

12. Использование вариантов выбора (CHOICES)

При использовании вариантов выбора рекомендуется:

- хранить в базе данных строки вместо чисел (хотя это не лучший вариант с точки зрения использования необязательной базы данных, на практике он удобнее, так как строки более наглядны, что позволяет использовать четкие фильтры с опциями **get** из коробки в **REST-**фреймворках).
- переменные для хранения вариантов являются константами. Поэтому они должны быть указаны в верхнем регистре.
- Указывайте варианты перед списками полей.
- если это список статусов, указывайте их в хронологическом порядке (например, **new**, **in_progress**, **completed**).
- вы можете использовать **Choices** из библиотеки **model_utils.** Возьмем, к примеру, модель **Article:**

```
from model_utils import Choices

class Article(models.Model):
    STATUSES = Choices(
            (0, 'draft', _('draft')),
            (1, 'published', _('published'))   )
    status = models.IntegerField(choices=STATUSES, default=STATUSES.draft)
    ...
```

13. Зачем нужен дополнительный .all()?

Используя **ORM,** не добавляйте дополнительный вызов метода **all** перед **filter(), count()** и т.д.

14. Много флагов в модели?

Если это обосновано, замените несколько **BooleanFields** на одно поле, подобное статусу. Например:

```
class Article(models.Model):
    is_published = models.BooleanField(default=False)
    is_verified = models.BooleanField(default=False)
    ...
```

Предположим, что логика нашего приложения предполагает, что статья изначально не публикуется и не проверяется, затем она проверяется и помечается **is_verified** в **True,** а затем публикуется. Вы можете заметить, что статья не может быть опубликована без проверки. Таким образом, всего есть **3** условия, но с двумя логическими полями у нас нет **4** возможных вариантов, и вы должны убедиться, что нет статей с неправильными комбинациями условий логических полей. Вот почему использование одного поля статуса вместо двух булевых полей является лучшим вариантом:

```
class Article(models.Model):
    STATUSES = Choices('new', 'verified', 'published')
    status = models.IntegerField(choices=STATUSES, default=STATUSES.draft)
    ...
```

Этот пример может быть не очень наглядным, но представьте, что в вашей модели есть **3** или более таких булевых полей, и контроль валидации для этих комбинаций значений полей может быть действительно утомительным.

15. Дублирование имени модели в имени поля

Не добавляйте имена моделей к полям, если в этом нет необходимости, например, если в таблице **User** есть поле **user_status -** следует переименовать поле в **status,** если в этой модели нет других статусов.

16. Грязные данные не должны находиться в базе

Всегда используйте **PositiveIntegerField** вместо **IntegerField,** если это не бессмысленно, потому что "плохие" данные не должны попадать в базу. По той же причине всегда используйте **unique,unique_together** для логически уникальных данных и никогда не используйте **required=False** в каждом поле.

17. Получение самого раннего/самого позднего объекта

Вы можете использовать **ModelName.objects.earliest('created'/'earliest')** вместо **order_by('created')[0],** а также поместить **get_latest_by** в **Meta model.** Следует помнить**,** что **latest/earliest,** а также **get** могут вызвать исключение **DoesNotExist.** Поэтому наиболее полезным вариантом является **order_by('created').first().**

18. Никогда не делайте len(queryset).

Не используйте **len** для получения количества объектов **queryset.** Для этой цели можно использовать метод **count.** Например**,** так**:** **len(ModelName.objects.all()),** сначала будет выполнен запрос на выборку всех данных из таблицы**,** затем эти данные будут преобразованы в объект **Python,** а длина этого объекта будет найдена с помощью **len.** Настоятельно рекомендуется не использовать этот метод**,** так как **count** будет обращаться к соответствующей **SQL-**функции **COUNT(). С помощью count будет проще выполнить запрос в этой базе данных и потребуется меньше ресурсов для выполнения кода python.**

19. If queryset - плохая идея

Не используйте **queryset** как булево значение**:** вместо **if queryset: do something** используйте **if queryset.exists(): do something.** Помните**,** что кверисеты ленивы**,** и если вы используете кверисет как булево значение**,** будет выполнен неподходящий запрос к базе данных**.**

20. Использование help_text в качестве документации

Использование **help_text** в полях модели в качестве части документации определенно облегчит понимание структуры данных вами**,** вашими коллегами и пользователями**-**администраторами**.**

21. Хранение информации о количестве денег

Не используйте **FloatField** для хранения информации о количестве денег**.** Вместо этого используйте для этой цели **DecimalField.** Вы также можете хранить эту информацию в центах**,** единицах и т.д**.**

22. Не используйте null=true, если вам это не нужно

null=True - Позволяет столбцу хранить нулевое значение.
blank=True - Будет использоваться только в том случае, если формы предназначены для валидации и не связаны с базой данных.
В текстовых полях лучше сохранять значение по умолчанию.

blank=True
default=''

Таким образом, вы получите только одно возможное значение для столбцов без данных.

23. Удалите _id

Не добавляйте суффикс **_id** к **ForeignKeyField** и **OneToOneField.**

24. Определите __unicode__ или __str__.

Во все неабстрактные модели добавьте методы **__unicode__(python 2)** или **__str__(python 3).** Эти методы всегда должны возвращать строки.

25. Прозрачный список полей

Не используйте **Meta.exclude** для описания списка полей модели в **ModelForm.** Лучше использовать для этого **Meta.fields,** так как он делает список полей прозрачным. Не используйте **Meta.fields="__all__"** по той же причине.

26. Не складывайте все загруженные пользователем файлы в одну папку.

Иногда даже отдельной папки для каждого поля **FileField** будет недостаточно, если ожидается большое количество загруженных файлов. Хранение большого количества файлов в одной папке означает, что файловая система будет искать нужный файл медленнее. Чтобы избежать подобных проблем, вы можете сделать следующее:

```
def get_upload_path(instance, filename):
    return os.path.join('account/avatars/', now().date().strftime("%Y/%m/%d"), filename)

class User(AbstractUser):
    avatar = models.ImageField(blank=True, upload_to=get_upload_path)
```

27. Используйте абстрактные модели

Если вы хотите разделить некоторую логику между моделями, вы можете использовать абстрактные модели.

```
class CreatedatModel(models.Model):

    created_at = models.DateTimeField(

        verbose_name=u "Created at",

        auto_now_add=True

    )


    class Meta:

        abstract = True

class Post(CreatedatModel):
...

class Comment(CreatedatModel):
...
```

28. Использование пользовательского менеджера и QuerySet

Чем больше проект, над которым вы работаете, тем чаще вы повторяете один и тот же код в разных местах.

Чтобы сохранить код **DRY** и распределить бизнес-логику в моделях, вы можете использовать пользовательские менеджеры и **Queryset.**

Например. Если вам нужно получить количество комментариев для постов, из примера выше.

```
class CustomManager(models.Manager):
    def with_comments_counter(self):
        return self.get_queryset().annotate(comments_count=Count('comment_set'))
```

Теперь вы можете использовать:

```
posts = Post.objects.with_comments_counter()
posts[0].comments_count
```

Если вы хотите использовать этот метод в связке с другими методами **queryset,** вам следует использовать пользовательский **QuerySet:**

```
class CustomQuerySet(models.query.QuerySet):
    """
    Замена QuerySet, и добавление дополнительных методов к QuerySet.
    """
    def with_comments_counter(self):
        """
        Добавляет счетчик комментариев к набору запросов
        """
        return self.annotate(comments_count=Count('comment_set'))
```

Теперь вы можете использовать:

```
posts = Post.objects.filter(...).with_comments_counter()
posts[0].comments_count
```