Django Models API

Создание и изменение объектов

```
from blog.models import Category

# создание
c = Category(title="Perl")
c.save()
# или за один вызов
c = Category.objects.create(title="Python")

# изменение
c.title = "About Python"
c.save()
```

Создание объектов со связями

```
from blog.models import Category, Post, Tag

t = Tag.objects.create(title="easy")
c = Category.objects.create(title="Python")

p = Post(title="Intro", text="...", category=c)
p.save()
p.tags = Tag.objects.all()[:3] # [ Tag ]
p.save()
p.tags.add(t)
```

Загрузка объекта из базы

```
# по ключу
try:
    post = Post.objects.get(pk=3)
except Post.DoesNotExist:
    post = None

# по другому полю
try:
    post = Post.objects.get(name="Python")
except Post.MultipleObjectsReturned:
    post = None
```

Выборка нескольких объектов

```
all_posts = Post.objects.all()
first_three = Post.objects.all()[:3]

c = Category.objects.get(id=3)

python_posts = Post.objects.filter(category=c)

css_posts = Post.objects.filter(title__contains="css")
css_posts = css_posts.order_by("-rating")
css_posts = css_posts[10:20]
```

QuerySets

QuerySet

QuerySet - объекты, представляющие собой запрос к базе данных. Именно **запрос**, а не его результаты. QuerySet являются **ленивыми (lazy)** объектами. Это значит, что запрос осуществляется не в момент создания QuerySet, а в момент **итерации по нему**, либо вызова метода, возвращающего результат.

Chaining

```
posts = Post.objects                        # ModelManager
posts = posts.filter(title__match="CSS")    # QuerySet
posts = posts.exclude(category_id=7)        # QuerySet
posts = posts.order_by("rating")            # QuerySet
posts = posts.reverse()                     # QuerySet
posts = posts[0:10]                         # [ Post ] offset 0 limit 10
```

Методы QuerySet (chaining)

- `filter`, `exclude` - фильтрация, в SQL это `WHERE`
- `order_by` - сортировка
- `annotate` - выборка агрегатов, в SQL это `JOIN` и `GROUP BY`
- `values` - выборка отдельных колонок, а не объектов
- `distinct` - выборка уникальных значений
- `select_related`, `prefetch_related` - выборка из нескольких таблиц

select_related и prefetch_related

```
posts = Post.objects.select_related("category")
# SELECT * FROM blog_post p LEFT JOIN blog_category c ON c.id = p.category_id;

posts = Post.objects.prefetch_related("category")
# SELECT * FROM blog_post;
# Get category_id list
# SELECT * FROM blog_category WHERE id IN (1, 4, 5);
```

Методы QuerySet (результат)

- `create` - создание нового объекта
- `update` - обновление всех подходящих объектов
- `delete` - удаление всех подходящих объектов
- `count` - выборка количества `COUNT(*)`
- `get_or_create` - выборка объекта или его создание
- `update_or_create` - обновление объекта или его создание

Синтаксис условий в QuerySet

В методах `filter` и `exclude`:

- `field = value` - точное совпадение
- `field__contains = value` - суффикс оператора `LIKE`
- `field__isnull`, `field__gt`, `field__lte`
- `relation__field = value` - условие по связанной таблице
- `category__title__contains = "Perl"`

Названия полей и таблиц не могут содержать `__`!

ModelManager

ModelManager

В модели содержатся методы для работы с одним объектом (одной строкой). В **ModelManager** содержатся объекты для работы со множеством объектов. **ModelManager** «по-умолчанию» содержит все те же методы что **QuerySet** и используется для создания **QuerySet** объектов связанных с данной моделью.

ModelManager «по-умолчанию»

```
class Post(models.Model):
    title = models.CharField()
    # ....

posts = Post.objects                    # Manager
posts = Post.objects.all()              # QuerySet
posts = Post.objects.filter(id__gt=10) # QuerySet
```

Свой ModelManager

```
class PostManager(models.Manager):
    def best_posts(self):
        return self.filter(rating__gt=50)
    def published(self):
        return self.filter(published=True)
    def create_draft(self, **kwargs):
        kwargs["draft"] = True
        return self.create(**kwargs)

class Post(models.Model):
    ...
    objects = PostManager()

posts = Post.objects   # PostManager
```

RelatedManager

```
class Post(models.Model):
    # ...
    tags = models.ManyToManyField(Tag)

p1 = Post.objects.get(pk=1)
tags = p1.tags  # RelatedManager
```

RelatedManager связан с конкретным объектом Post и во все выборки будет добавлять условие `post=p1`

Методы RelatedManager

- `create(**kwargs)` - создание нового тэга, связанного с постом
- `add(tag)` - привязка существующего тэга `tag` к текущему посту
- `remove(tag)` - отвязка существующего тэга `tag` от текущего поста
- `clear()` - очистка списка тэгов у текущего поста

Миграции

Миграции

Миграция - это процедура изменения схемы базы данных для приведения ее в соответствие с моделями.

Начиная с версии 1.7 Django поддерживает миграции на уровне фреймворка.

Django миграции

- `./manage.py makemigrations` - анализ изменений в моделях и создание миграций.
- `./manage.py migrate` - применение новых миграций к базе данных.
  - \+ поддержка различных СУБД
  - \+ прямые и обратные миграции
  - − на практике часто неудобные или недостаточные

Свои миграции

project/migration/2015-08-08-more-post-fields.py

```
#!/usr/bin/python3
from django.core.management import setup_environ
from project import settings
setup_environ(settings)
from django.db import connection
cursor = connection.cursor()
cursor.execute("""
    alter table blog_post add column is_best tinyint(1)
""");
```

Best practices

Fat controller

Типичная проблема начинающих разработчиков - размещение логики в контроллерах. Это плохое решение, у которого есть имя - антипаттерн **Fat Controller**.

Размещение логики в контроллере лишает вас возможности использовать ее повторно. Всю бизнес-логику приложения следует размещать в **моделях** или в отдельном модуле **services.py**.