ТЕХНОАТОМ

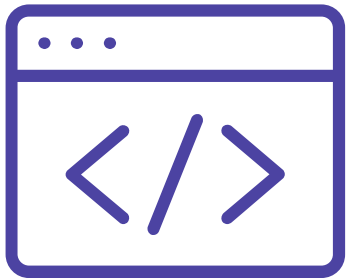
# Веб-программирование Python

## Лекция 4. Базы данных, модели Django | Часть 2

Михалев Олег

# Сегодня

- Формирование запросов
- Агрегация
- Менеджеры модели
- Транзакции



# Наследники Model - это основной источник данных

Они содержат набор полей и поведение

Для получения объектов из базы данных, через **Manager** модели создается экземпляр **QuerySet**

```
1.>>> Student.objects
2.<django.db.models.manager.Manager object at 0x006568B0>
3.>>> Student.objects.all()
4.[...]
```

Для фильтрации объектов применяются методы **filter** (совпадение по условию) и **exclude** (исключение по условию)

Результат фильтрации также является экземпляром **QuerySet**

Каждое новое изменение порождает новый экземпляр **QuerySet**

# Промежуточные значения обычно не сохраняют, работая с цепочками фильтров

```
1.>>> Student.objects
2....      .exclude(expelled=True)
3....      .filter(points__gt=0)
```

Создание нового экземпляра **QuerySet** не подразумевает запрос к СУБД

Реальный запрос осуществляется только в случае обращения к данным в **QuerySet**

**Обращение к СУБД происходит в случае:**

- Итерация, получение длины
- Срезы и обращение по позиции
- Приведение типов

```
>>> queryset = Student.objects.exclude(expelled=True)
>>> queryset = queryset.filter(points__gt=0).order_by('-points')

>>> top_student = queryset[0]
>>> print('Best result: %s' % top_student.name)

>>> print('Top 10 results:')
>>> for student in queryset[:10]:
...     print('₩t%s' % student.name)
...

>>> students = list(queryset)
>>> print(students[-1])
```

# В случае обращения к СУБД результаты кэшируются в **QuerySet**

Со срезами и обращениями по позиции необходимо пояснение

Полезно экономить память и получать из БД только те поля, которые нужны

**QuerySet** позволяет ограничить список полей методами **defer** и **only**

# Метод **defer** исключает из выборки указанные поля

```
1.>>> Student.objects.defer('first_name', 'last_name')
```

Метод **only** исключает из выборки все поля кроме указанных

```
1.>>> Student.objects.only('last_name')
```

При этом следует помнить, что отдельные экземпляры моделей также имеют кэш

Если поле исключено, то обращение к нему в первый раз инициирует повторный запрос

Часто при чтении объектов, чтобы избежать неоднозначности с отложенными полями (а еще более часто - чтобы избежать создания тяжелых экземпляров модели), используют простые структуры данных

**QuerySet** позволяет использовать простые структуры данных для хранения результата

**QuerySet** предоставляет методы **values** и **values_list**

Метод **values** позволяет конструировать **QuerySet**, который возвращает словари вместо моделей. Имена выбираемых полей можно передать в качестве аргументов

```
>>> Student.objects.filter(last_name='sparrow').values()
[
    {
        'first_name:' 'jack',
        'last_name': 'sparrow',
        'points': 42.00
    }
]
```

Метод **values_list** позволяет конструировать **QuerySet**, который возвращает кортежи. Порядок полей в кортеже соответствует порядку полей указанных в качестве аргументов (или согласно порядку объявления их в модели)

```
1. >>> Student.objects.values_list('first_name', 'last_name')
2. [
3.     ('jack', 'sparrow'),
4.     ...
5. ]
```

Если необходимо получить одно поле, метод **values_list** может создать плоскую структуру - список

```
1. >>> Student.objects.values_list('points', flat=True)
2. [42.0, 34.5, ...]
```

# Методы **values** и **values_list** позволяют обращаться к полям связанных моделей через нотацию **lookup**

```
>>> Student.objects.filter(login='sparrow')
...     .values('login', 'course__name', 'points')
[
    {
        'login': 'sparrow',
        'course__name:' 'python programming',
        'points': 42.00
    }
]
```

Lookup по полям связанных полей всегда генерирует JOIN в SQL-запросе

JOIN в SQL - реализация операция соединения

**INNER JOIN**

внутреннее соединение

**OUTER JOIN**

внешнее  соединение

**CROSS JOIN**

декартово произведение

## Cources

| ID | Name |
|---|---|
| 1 | Python |
| 2 | C++ |
| 3 | Java |

## Students

| Name | CourseID |
|---|---|
| Д'Артаньян | 1 |
| Атос | 2 |
| Портос | 2 |
| Арамис | 4 |

# INNER JOIN
выбираются только полностью совпадающие данные

```
1. SELECT
2.     Students.Name AS Name,
3.     Students.CourseID AS CourseID,
4.     Courses.Name AS CourseName
5. FROM Students
6. INNER JOIN Cources
7.     ON (Students.CourseID = Course.ID);
```

**Students**

| Name | CourseID |
|---|---|
| Д'Артаньян | 1 |
| Атос | 2 |
| Портос | 2 |
| Арамис | 4 |

**Cources**

| ID | Name |
|---|---|
| 1 | Python |
| 2 | C++ |
| 3 | Java |

| Name | CourseID | CourseName |
|---|---|---|
| Д'Артаньян | 1 | **Python** |
| Атос | 2 | **C++** |
| Портос | 2 | **C++** |

**LEFT OUTER JOIN**

**RIGHT OUTER JOIN**

**FULL OUTER JOIN**

# LEFT OUTER JOIN

совпадения подставляются в левую часть

```
1. SELECT
2.     Students.Name AS Name,
3.     Students.CourseID AS CourseID,
4.     Courses.Name AS CourseName
5. FROM Students
6. LEFT OUTER JOIN Cources
7.     ON (Students.CourseID = Course.ID);
```

**Students**

| Name | CourseID |
|---|---|
| Д'Артаньян | 1 |
| Атос | 2 |
| Портос | 2 |
| Арамис | 4 |

**Cources**

| ID | Name |
|---|---|
| 1 | Python |
| 2 | C++ |
| 3 | Java |

| Name | CourseID | CourseName |
|---|---|---|
| Д'Артаньян | 1 | **Python** |
| Атос | 2 | **C++** |
| Портос | 2 | **C++** |
| Арамис | 4 | **NULL** |

# RIGHT OUTER JOIN

совпадения подставляются в правую часть

```
1. SELECT
2.     Students.Name AS Name,
3.     Students.CourseID AS CourseID,
4.     Courses.Name AS CourseName
5. FROM Students
6. RIGHT OUTER JOIN Cources
7.     ON (Students.CourseID = Course.ID);
```

**Students**

| Name | CourseID |
|---|---|
| Д'Артаньян | 1 |
| Атос | 2 |
| Портос | 2 |
| Арамис | 4 |

**Cources**

| ID | Name |
|---|---|
| 1 | Python |
| 2 | C++ |
| 3 | Java |

| Name | CourseID | CourseName |
|---|---|---|
| Д'Артаньян | 1 | **Python** |
| Атос | 2 | **C++** |
| Портос | 2 | **C++** |
| NULL | NULL | **Java** |

# FULL OUTER JOIN
## полное внешние соединение

```
1. SELECT
2.     Students.Name AS Name,
3.     Students.CourseID AS CourseID,
4.     Courses.Name AS CourseName
5. FROM Students
6. FULL OUTER JOIN Cources
7.     ON (Students.CourseID = Course.ID);
```

**Students**

| Name | CourseID |
|---|---|
| Д'Артаньян | 1 |
| Атос | 2 |
| Портос | 2 |
| Арамис | 4 |

**Cources**

| ID | Name |
|---|---|
| 1 | Python |
| 2 | C++ |
| 3 | Java |

| Name | CourseID | CourseName |
|---|---|---|
| Д'Артаньян | 1 | **Python** |
| Атос | 2 | **C++** |
| Портос | 2 | **C++** |
| Арамис | 4 | **NULL** |
| NULL | NULL | **Java** |

CROSS JOIN не имеет собственных условий, но фильтруется выражением WHERE

Каждая строка одной таблицы соединяется с каждой строкой второй таблицы

Django позволяет "предвыбирать" связанные сущности, для этого используются методы **select_related** и **prefetch_related**

Метод **select_related** выберет связанную модель, объявленную полем **ForeignKey** или **OneToOneField**

```
1. >>> Student.objects.exclude(expelled=True)
2. ...     .select_related('course')
```

# Метод **prefetch_related** выберет множество связанных сущностей

```
1. >>> Course.objects.all().prefetch_related('students')
```

А как работать с этим множеством?

Простейшим примером агрегирующих операций являются методы **count** (возвращает количество записей выборки) и **exists** (возвращает истину, если существует хотя бы одна запись)

Более сложные операции представлены классами модуля **django.db.models** (Count, Avg, Max, Min, Sum и т.д.) и используются методами **aggregate** и **annotate**

Метод **aggregate** позволяет получить словарь агрегированных значений

```
1.>>> Student.objects.aggregate(avg_points=Avg('points'))
2.{'avg_points': 93.23}
```

Метод **annotate** добавит к каждому объекту из результата **QuerySet** аннотации

```
1.>>> courses = Cource.objects.annotate(Count('students'))
2.>>> cources[0].students__count
3.4
```

При передаче именованных параметров у экземпляра модели из результата появится соответствующее имя

```
1. >>> courses = Cource.objects.annotate(members=Count('students'))
2. >>> cources[0].members
3. 4
```

Несложно заметить, что **annotate** удобен для вычисления значений агрегирующих функций по множеству связанных сущностей

# Последующие за **annotate** фильтры позволяют скорректировать выборку

```
1. >>> Cource.objects.exclude(start_date__lt=today)
2. ...     .annotate(points=Sum('students__points'))
3. ...     .filter(points__gt=0)
```

Важно помнить, что порядок следования **filter/exclude** и **annotate** в цепочке имеет значение - **filter/exclude** перед **annotate** имеют влияние на само агрегированное значение

```
1. SELECT
2.     Courses.Name,
3.     AVG(Students.Points) AS Points
4. FROM Cources
5. LEFT OUTER JOIN Students
6.     ON (Cources.ID = Students.CourseID)
7. WHERE
8.     Points > 0
9. GROUP BY Cources.Name;
```

# Возможно указывать поля группировки

```
>>> Cource.objects.values('start_date', 'end_date')
...     .annotate(
...         season_points=Sum('students__points')
...     )
```

Основные методы **QuerySet** вызываемые на моделях мы уже знаем

Иногда полезно определять свои

TEXHOATOM

```
1. class StudentQuerySet(models.QuerySet):

2.     def achievers(self):
3.         return self.filter(points__gt=95)


4. class StudentManager(models.Manager):

5.     def get_queryset(self):
6.         return StudentQuerySet(self.model, using=self._db)
7.             .exclude(expelled=True)

8.     def achievers(self):
9.         return self.get_queryset().achievers()
```

```
1. class Student(models.Model):
2.     ...

3.     objects = StudentManager()

4.     # StudentQuerySet.as_manager()
```

# И напоследок о транзакциях

```
1. class StudentQuerySet(models.QuerySet):

2.     def achievers(self):
3.         return self.filter(points__gt=95)


4. class StudentManager(models.Manager):

5.     def get_queryset(self):
6.         return StudentQuerySet(self.model, using=self._db)
7.             .exclude(expelled=True)

8.     def achievers(self):
9.         return self.get_queryset().achievers()
```

TEXHOATOM

```
1. class Account(models.Model):

2.     cash_back = models.ForeignKey(CashBack, related_name='+')

3.     def make_charge(self, value):
4.         charge = Charge.objects.create(value, account=self)
5.         if value < 0:
6.             value += self.cash_back.coefficient * abs(value)
7.         self.total += value
8.         self.save()
```

# Проблемы можно предотвратить используя транзакции

```
1. from django.db import transaction

2. try:
3.     ... # make actions
4.     transaction.commit()
5. except:
6.     transaction.rollback()
```

# Удобнее использовать менеджер контекста и **atomic**

```
1. from django.db import transaction

2. with transaction.atomic():
3.     ... # make actions
```

# Еще удобнее использовать декораторы и **atomic**

```
1. class Account(models.Model):

2.     total = models.DecimalField(default=0)

3.     @transaction.atomic(savepoint=False)
4.     def make_charge(self, value):
5.         charge = Charge.objects.create(value, account=self)
6.         self.total += value
7.         self.save()
```


# Спасибо за внимание!

Михалев Олег
mailto:mhalairt@gmail.com

# Домашнее задание

ТЕХНОАТОМ

Если это необходимо - использовать транзакции СУБД при изменении баланса счета и создании списания/зачисления на счет. Добавить в приложение представление статистики, где выводить изменения баланса в сумме по месяцам.

# Полезные ссылки

ТЕХНОАТОМ

**Django QuerySets**
https://docs.djangoproject.com/en/1.10/ref/models/querysets/

**Django Aggregation**
https://docs.djangoproject.com/en/1.10/topics/db/aggregation/

**Django Database transactions**
https://docs.djangoproject.com/en/1.10/topics/db/transactions/