Материалы занятия

Курс: Web-разработка

Дисциплина: Создание web-приложений с использованием фреймворка Django

Тема занятия № 31: Модуль 16. Модели: расширенные инструменты

1. СОЗДАНИЕ СВОИХ ДИСПЕТЧЕРОВ ЗАПИСЕЙ

Диспетчер записей — это объект, предоставляющий доступ к набору записей, которые хранятся в модели. По умолчанию он представляет собой экземпляр класса Manager ИЗ МОДУЛЯ dj ango. Db.modeis И хранится В атрибуте objects модели.

Создание диспетчеров записей

Диспетчеры записей наследуются от класса Manager. В них можно как переопределять имеющиеся методы, так и объявлять новые.

Переопределять имеет смысл только метод get_queryset(self), который должен возвращать набор записей текущей модели в виде экземпляра класса Queryset из модуля django.db.models. Обычно в теле переопределенного метода сначала вызывают тот же метод базового класса, чтобы получить изначальный набор записей, устанавливают у него фильтрацию, сортировку, добавляют вычисляемые поля и возвращают в качестве результата.

Код диспетчера записей rubricmanager, который возвращает набор рубрик уже отсортированным ПО ПОЛЯМ order И name. Помимо того, он объявляет дополнительный метод order_by_bb_count (), который возвращает набор рубрик, отсортированный по убыванию количества относящихся к ним объявлений.

```
from django.db import models

class RubricManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().order_by('order', 'name')

    def order_by_bb_count(self):
        return super().get_queryset().annotate(
                        cnt=models.Count('bb')).order_by('-cnt')
```

Использовать новый диспетчер записей в модели можно трояко:

□ в качестве единственного диспетчера записей — объявив в классе модели атрибут objects и присвоив ему экземпляр класса диспетчера записей:

```
class Rubric(models.Model):
    . . .
    objects = RubricManager()
```

Теперь, обратившись к атрибуту objects модели, мы получим доступ к нашему диспетчеру:

```
>>> from bboard.models import Rubric
>>> # Получаем набор записей, возвращенный методом get_queryset()
>>> Rubric.objects.all()
```

```
<QuerySet [<Rubric: Транспорт>, <Rubric: Недвижимость>,
<Rubric: Мебель>, <Rubric: Бытовая техника>, <Rubric: Сантехника>,
<Rubric: Растения>, <Rubric: Сельхозинвентарь>]>
>>> # Получаем набор записей, возвращенный вновь добавленным методом
>>> # order_by_bb_count()
>>> Rubric.objects.order_by_bb_count()
<QuerySet [<Rubric: Недвижимость>, <Rubric: Транспорт>,
<Rubric: Бытовая техника>, <Rubric: Сельхозинвентарь>,
<Rubric: Мебель>, <Rubric: Сантехника>, <Rubric: Растения>]>
```

□ то же самое, только с использованием атрибута класса с другим именем:

```
class Rubric(models.Model):
    . . .
    bbs = RubricManager()
. . .
>>> # Теперь для доступа к диспетчеру записей используем атрибут
>>> # класса bbs
>>> Rubric.bbs.all()
```

□ в качестве дополнительного диспетчера записей — присвоив его другому атрибуту класса модели:

```
class Rubric(models.Model):
    . . .
    objects = models.Manager()
    bbs = RubricManager()
```

Теперь в атрибуте objects хранится диспетчер записей, применяемый по умолчанию, а в атрибуте bbs — наш диспетчер записей. И мы можем пользоваться сразу двумя диспетчерами записей. Пример:

```
>>> Rubric.objects.all()
<QuerySet [<Rubric: Бытовая техника>, <Rubric: Мебель>,
<Rubric: Недвижимость>, <Rubric: Растения>, <Rubric: Сантехника>,
<Rubric: Сельхозинвентарь>, <Rubric: Транспорт>]>
>>> Rubric.bbs.all()
<QuerySet [<Rubric: Транспорт>, <Rubric: Недвижимость>,
<Rubric: Мебель>, <Rubric: Бытовая техника>, <Rubric: Сантехника>,
<Rubric: Растения>, <Rubric: Сельхозинвентарь>]>
```

Здесь нужно учитывать один момент. Первый объявленный в модели диспетчер записей будет рассматриваться Django как используемый по умолчанию, применяемый при выполнении различных служебных задач (в нашем случае это диспетчер записей Manager, присвоенный атрибуту objects). Поэтому ни в коем случае нельзя задавать в таком диспетчере данных фильтрацию, иначе записи модели, не удовлетворяющие ее критериям, окажутся необработанными.

На заметку!

Можно дать атрибуту, применяемому для доступа к диспетчеру записей, другое имя без задания для него нового диспетчера записей:

```
class Rubric(models.Model):
    . . .
    bbs = models.Manager()
```

Создание диспетчеров обратной связи

Аналогично можно создать свой диспетчер обратной связи, который выдает набор записей вторичной модели, связанный с текущей записью первичной модели.

Его класс также объявляется как производный от класса Manager из модуля Django.db.models и также указывается в модели присваиванием его экземпляра атрибуту класса модели.

Код диспетчера обратной связи вьмападег, возвращающий связанные объявления отсортированными по возрастанию цены.

```
from django.db import models

class BbManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().order_by('price')
```

Чтобы из записи первичной модели получить набор связанных записей вторичной модели с применением нового диспетчера обратной связи, придется явно указать этот диспетчер. Для этого у объекта первичной записи вызывается метод, имя которого совпадает с именем атрибута, хранящего диспетчер связанных записей.
Этому методу передается параметр manager, в качестве значения ему присваивается строка с именем атрибута класса вторичной модели, которому был присвоен объект нового диспетчера. Метод вернет в качестве результата набор записей, сформированный этим диспетчером.
Так, указать диспетчер обратной связи вьмападег в классе модели вь можно следующим образом (на всякий случай не забыв задать диспетчер, который будет использоваться по умолчанию):

```
class Bb(models.Model):
    . . .
    objects = models.Manager()
    by_price = BbManager()
```

Проверим его в деле:

```
>>> from bboard.models import Rubric
>>> r = Rubric.objects.get(name='Недвижимость')
>>> # Используем диспетчер обратной связи по умолчанию. Объявления будут
>>> # выведены отсортированными по умолчанию — по убыванию даты
>>> # их публикации
>>> r.bb_set.all()
```

```
<QuerySet [<Bb: Bb object (6)>, <Bb: Bb object (3)>,
<Bb: Bb object (1)>]>
>>> # Используем свой диспетчер обратной связи. Объявления сортируются
>>> # по возрастанию цены
>>> r.bb_set(manager='by_price').all()
<QuerySet [<Bb: Bb object (6)>, <Bb: Bb object (1)>,
<Bb: Bb object (3)>]>
```

2. СОЗДАНИЕ СВОИХ НАБОРОВ ЗАПИСЕЙ

Еще можно объявить свой класс набора записей, сделав его производным от класса Queryset из модуля dj ango. Db. Mode is. Объявленные в нем методы могут фильтровать и сортировать записи по часто встречающимся критериям, выполнять в них часто применяемые агрегатные вычисления и создавать часто используемые вычисляемые поля.

Код набора записей rubricqueryset с дополнительным методом, вычисляющим количество объявлений, имеющихся в каждой рубрике, и сортирующим рубрики по убыванию этого количества.

```
from django.db import models

class RubricQuerySet(models.QuerySet):
    def order_by_bb_count(self):
        return self.annotate(cnt=models.Count('bb')).order_by('-cnt')
```

Для того чтобы модель возвращала набор записей, представленный экземпляром объявленного нами класса, понадобится также объявить свой диспетчер записей (как это сделать, было рассказано ранее). Прежде всего, в методе Get queryset () он сформирует и вернет в качестве результата экземпляр нового класса набора записей. Конструктору этого класса в качестве первого позиционного параметра следует передать используемую модель, которую можно извлечь из атрибута model, а в качестве параметра using — базу данных, в которой хранятся записи модели и которая извлекается из атрибута db.

Помимо этого, нужно предусмотреть вариант, когда объявленные в новом наборе записей дополнительные методы вызываются не у набора записей:

```
rs = Rubric.objects.all().order_by_bb_count()
```

А непосредственно у диспетчера записей:

```
rs = Rubric.objects.order_by_bb_count()
```

Для этого придется объявить одноименные методы еще и в классе набора записей и выполнять в этих методах вызовы соответствующих им методов набора записей.

Код диспетчера записей rubricmanager, призванного обслуживать набор записей rubricqueryset.

```
from django.db import models

class RubricManager(models.Manager):
    def get_queryset(self):
        return RubricQuerySet(self.model, using=self._db)

    def order_by_bb_count(self):
        return self.get_queryset().order_by_bb_count()
```

Новый диспетчер записей указывается в классе модели:

```
class Rubric(models.Model):
    . . .
    objects = RubricManager()
```

Проверим созданный набор записей в действии (вывод пропущен ради краткости):

```
>>> from bboard.models import Rubric
>>> Rubric.objects.all()
. . .
>>> Rubric.objects.order_by_bb_count()
. . .
```

Писать свой диспетчер записей только для того, чтобы ''подружить" модель с новым набором записей, вовсе не обязательно. Можно использовать одну из двух фабрик классов, создающих классы диспетчеров записей на основе наборов записей и реализованных в виде методов:

□ as manager () — вызывается у класса набора записей и возвращает обслуживающий его экземпляр класса диспетчера записей:

```
class Rubric(models.Model):
    . . .
    objects = RubricQuerySet.as_manager()
```

□ from queryset (<класс набора записей>) — вызывается у класса диспетчера записей и возвращает ссылку на производный класс диспетчера записей, обслуживающий заданный набор записей:

```
class Rubric(models.Model):
    . . .
    objects = models.Manager.from_queryset(RubricQuerySet)()
```

Можно сказать, что обе фабрики классов создают новый класс набора записей и переносят в него методы из базового набора записей. В обоих случаях действуют следующие правила переноса методов:

□ обычные методы переносятся по умолчанию;

□ псевдочастные методы (имена которых предваряются символом подчеркивания) не переносятся по умолчанию;

□ записанный у метода атрибут queryset_oniy со значением False указывает перенести метод;

□ записанный у метода атрибут queryset_oniy со значением True указывает не переносить метод.

Пример:

```
class SomeQuerySet(models.QuerySet):
    # Этот метод будет перенесен
    def method1(self):
        . . .
    # Этот метод не будет перенесен
    def _method2(self):
        . . .

    # Этот метод будет перенесен
    def _method3(self):
        . . .
    _method3.queryset_only = False

    # Этот метод не будет перенесен
    def method4(self):
        . . .
    _method4.queryset_only = True
```

3. УПРАВЛЕНИЕ ТРАНЗАКЦИЯМИ

Django предоставляет удобные инструменты для управления транзакциями, которые пригодятся при программировании сложных решений.

Автоматическое управление транзакциями

Проще всего активизировать автоматическое управление транзакциями, при котором Django самостоятельно запускает транзакции и завершает их— с подтверждением, если все прошло нормально, или с откатом, если в контроллере возникла ошибка.

Внимание!

Автоматическое управление транзакциями работает только в контроллерах. В других модулях (например, посредниках) управлять транзакциями придется вручную.

Автоматическое управление транзакциями в Django может функционировать в двух режимах.

Режим по умолчанию: каждая операция — в отдельной транзакции в этом режиме каждая отдельная операция с моделью — чтение, добавление, правка и удаление записей — выполняется в отдельной транзакции.
Чтобы активировать этот режим, следует задать такие настройки базы данных:
□ параметру atomic request— дать значение False (или вообще удалить этот параметр, поскольку False — его значение по умолчанию). Тем самым мы предпишем выполнять каждую операцию с базой данных в отдельной транзакции;
□ параметру autocommit — дать значение True (или вообще удалить этот параметр, поскольку True — его значение по умолчанию). Так мы включим автоматическое завершение транзакций по окончании выполнения контроллера.
Собственно, во вновь созданном проекте Django база данных изначально настроена на работу в этом режиме.
Режим по умолчанию подходит для случаев, когда в контроллере выполняется не более одной операции с базой данных. В простых сайтах наподобие нашей доски объявлений обычно так и бывает.
На заметку!
Начиная с Django 2.2, операция с базой данных, которая может быть выполнена в один запрос, не заключается в транзакцию. Это сделано для повышения производительности.

Режим атомарных запросов
В этом режиме все операции с базой данных, происходящие в контроллере (т. е. на протяжении одного HTTP-запроса), выполняются в одной транзакции.
Переключить Django-сайт в такой режим можно, задав следующие настройки у базы данных:
□ параметру atomic request — дать значение True, чтобы, собственно, включить режим атомарных запросов;
□ параметру autocommit — дать значение True (или вообще удалить этот параметр).
Пример:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
        'ATOMIC_REQUEST': True,
    }
}
```

Этот режим следует активировать, если в одном контроллере выполняется сразу несколько операций, изменяющих данные в базе. Он гарантирует, что или в базу будут внесены все требуемые изменения, или, в случае возникновения нештатной ситуации, база останется в своем изначальном состоянии.

Режим по умолчанию на уровне контроллера
Если на уровне базы данных включен режим атомарных запросов, то на уровне какого-либо контроллера-функции можно включить режим управления транзакциями по умолчанию (в

котором отдельная операция с базой данных выполняется в отдельной транзакции). Для этого достаточно указать перед контроллером-функцией декоратор non_atomic_requests () ИЗ модуля django.db. Transaction. Пример:

```
from django.db import transaction

@transaction.non_atomic_requests
def my_view(request):
    # В этом контроллере действует режим обработки транзакций по умолчанию
```

Режим атомарных запросов на уровне контроллера

Аналогично, если на уровне базы данных действует режим по умолчанию, то на уровне какого-либо контроллера-функции можно включить режим атомарных запросов. Для этого применяется функция atomic) [savepoint=True]) из модуля Django.db.transaction. Ее можно использовать как:

О декоратор, указываемый перед контроллером-функцией:

```
from django.db.transaction import atomic

@atomic
def edit(request, pk):
    # В этом контроллере будет действовать режим атомарных запросов
    . . .
```

□ менеджер контекста в блоке with, в содержимом которого нужно включить режим атомарных запросов:

```
if formset.is_valid():
    with atomic():
        # Выполняем сохранение всех форм набора в одной транзакции
    return redirect('bboard:index')
```

Допускаются вложенные блоки with:

```
if formset.is_valid():
    with atomic():
        for form in formset:
            if form.cleaned_data:
                with atomic():
                    # Выполняем сохранение каждой формы набора
                    # в отдельном вложенном блоке with
```

В этом случае при входе во внешний блок with будет, собственно, запущена транзакция, а при входе во вложенный блок with— создана точка сохранения.
При выходе из вложенного блока выполняется подтверждение точки сохранения (если все прошло успешно) или же откат до состояния на момент ее создания (в случае возникновения ошибки). Наконец, после выхода из внешнего блока With происходит подтверждение или откат самой транзакции.
Каждая созданная точка сохранения отнимает системные ресурсы. Поэтому предусмотрена возможность отключить их создание, для чего достаточно в вызове функции atomic о указать параметр savepoint со значением False.

Ручное управление транзакциями
Если активно ручное управление транзакциями, то запускать транзакции, как и при автоматическом режиме, будет фреймворк, но завершать их нам придется самостоятельно.
Чтобы активировать ручной режим управления транзакциями, нужно указать следующие настройки базы данных:
□ параметру atomic request — дать значение False (если каждая операция с базой данных должна выполняться в отдельной транзакции) или True (если все операции с базой данных должны выполняться в одной транзакции);
□ параметру autocommit — дать значение False, чтобы отключить автоматическое завершение транзакций.
Для ручного управления транзакциями применяются следующие функции из модуля django.db.transaction:
□ commit () — завершает транзакцию с подтверждением;
□ rollback () — завершает транзакцию с откатом;
О savepoint () — создает новую точку сохранения и возвращает ее идентификатор в качестве результата;
□ savepoint_commit (<идентификатор точки сохранения^ — выполняет подтверждение точки сохранения с указанным идентификатором;
□ savepoint_rollback (Сидентификатор точки сохранения^ — выполняет откат ДО ТОЧКИ сохранения С указанным идентификатором;
□ clean savepoints () — сбрасывает счетчик, применяемый для генерирования уникальных идентификаторов точек сохранения;
□ get autocommit () — возвращает True, если для базы данных включен режим
Автоматического завершения транзакции, и False — в противном случае;

□ set autocommit (<режим>) — включает или отключает режим автоматического завершения транзакции для базы данных. Режим указывается в виде логической величины: True включает автоматическое завершение транзакций, False — отключает.
Пример ручного управления транзакциями при сохранении записи:

```
from django.db import transaction
. . .
if form.is_valid():
    try:
        form.save()
        transaction.commit()
    except:
        transaction.rollback()
```

Пример ручного управления транзакциями при сохранении набора форм с использованием точек сохранения:

```
if formset.is_valid():
    for form in formset:
        if form.cleaned_data:
            sp = transaction.savepoint()
            try:
                form.save()
                transaction.savepoint_commit(sp)
            except:
                transaction.savepoint_rollback(sp)
            transaction.commit()
```

Обработка подтверждения транзакции
Существует возможность обработать момент подтверждения транзакции. Для этого достаточно вызвать функцию on_commit(<функция-обработчик>) ИЗ МОДУЛЯ django.
Db.transaction, передав ей ссылку на функцию-обработчик. Последняя не должна ни принимать параметров, ни возвращать результат. Пример:

```
from django.db import transaction

def commit_handler():
    # Выполняем какие-либо действия после подтверждения транзакции

transaction.on_commit(commit_handler)
```

Указанная функция будет вызвана только после подтверждения транзакции, но не после ее отката, подтверждения или отката точки сохранения. Если в момент вызова функции on commit () активная транзакция отсутствует, то функция-обработчик выполнена не будет.