



Урок 8

Продвинутая работа с Django-ORM.

Тестирование в Django

Работа с атрибутами модели на уровне БД. Объекты F и Q.
Реализация сложной логики при помощи Case и When. Класс TestCase – основа тестов в Django. Smoke-тестирование.
Тестирование приложения аутентификации. Тест контроллеров

[Введение](#)

[Продвинутые запросы в Django-ORM](#)

[Эффективное обновление атрибута нескольких объектов при помощи «.update\(\)»](#)

[Объект «F» - обновление полей без загрузки значений](#)

[Объект «F» - задаем скидку на товары в категории за один запрос](#)

[Объект «Q» - логика в запросах](#)

[«Conditional-expressions» - логика на уровне QuerySet](#)

[Тестирование в Django](#)

[Тестирование работоспособности](#)

[Тестирование аутентификации пользователя](#)

[Тестирование методов моделей](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Введение

Итак, на курсе мы довели наш проект до уровня Production. На последнем занятии познакомимся с несколькими полезными инструментами и научимся писать тесты на основе модуля «unittest». Особенно это полезно при рефакторинге кода и дальнейшей оптимизации производительности, так как позволяет убедиться, что все работает по-прежнему.

Продвинутые запросы в Django-ORM

Эффективное обновление атрибута нескольких объектов при помощи «.update()»

При удалении категории товаров в админке, мы задаем значение False для атрибута «is_active», но для товаров категории этот атрибут не меняем. В каталоге все хорошо: неактивная категория не видна, и товары не отображаются. Когда будем редактировать заказ, то обнаружим баг: товары удаленной категории добавляются в заказ. Можно, конечно, фильтровать продукты еще и по атрибуту категории «is_active» при создании формы набора. Но мы отметим продукты неактивными при помощи метода «[.update\(\)](#)» объекта QuerySet:

geekshop/adminapp/views.py

```
...
from django.dispatch import receiver
from django.db.models.signals import pre_save
from django.db import connection
...
def db_profile_by_type(prefix, type, queries):
    update_queries = list(filter(lambda x: type in x['sql'], queries))
    print(f'db_profile {type} for {prefix}:')
    [print(query['sql']) for query in update_queries]

@receiver(pre_save, sender=ProductCategory)
def product_is_active_update_productcategory_save(sender, instance, **kwargs):
    if instance.pk:
        if instance.is_active:
            instance.product_set.update(is_active=True)
        else:
            instance.product_set.update(is_active=False)

    db_profile_by_type(sender, 'UPDATE', connection.queries)
```

Вместо работы с методами классов «ProductCategoryUpdateView()» и «ProductCategoryDeleteView()» мы воспользовались механизмом сигналов - так будет меньше кода. Атрибут «product_set Django» создан автоматически для связи с моделью Product по внешнему ключу. В этом атрибуте получаем QuerySet, который позволяет получить все продукты в данной категории и применяем к нему метод «.update()».

Внимание: этот метод не доступен для модели - только для QuerySet. Также следует помнить, что он не вызывает сигналы «pre_save» и «post_save» для моделей продуктов, ведь выполняется один запрос, а не несколько.

Для проверки импортируем объект соединения с БД «connection» из модуля «django.db». Его атрибут «queries» содержит словари с кодом и временем SQL-запросов, которые были выполнены в контроллере. Создаем функцию «db_profile_by_type()», которая отфильтровывает запросы определенного типа (например, «UPDATE», «DELETE», «SELECT», «INSERT INTO») и выводит их текст в консоль:

```
UPDATE "mainapp_product"
SET "is_active" = 1
WHERE "mainapp_product"."category_id" = 4
```

Видим, что Django справился с задачей идеально - все произошло на уровне базы данных.

Объект «F» - обновление полей без загрузки значений

Продолжим работу на низком уровне, но уже с отдельными объектами моделей.

Классическим вариантом обновления значения атрибута модели является следующая цепочка: выполнение запроса для получения текущего значения, его изменение и выполнение запроса для сохранения обновленного значения. Именно так у нас в проекте реализовано добавление товаров в корзину:

geekshop/basketapp/views.py

```
...
def basket_add(request, pk):
    ...
    old_basket_item = Basket.get_product(user=request.user, product=product)
    if old_basket_item:
        old_basket_item[0].quantity += 1
        old_basket_item[0].save()
    else:
        ...
    ...
```

Зачастую нет необходимости дорогостоящего преобразования объекта из представления в БД в представление на уровне python-объекта. Например, при инкременте или декременте числового значения поля. Есть смысл выполнить действие прямо на уровне БД. Для этого из модуля «django.db.models» импортируем [F-объект](#):

```
old_basket_item[0].quantity = F('quantity') + 1
```

Этот объект позволяет пробросить значение атрибута модели из базы данных в выражение (вместо извлечения этого значения). Можно провести аналогию с указателями в программировании. Имя класса ассоциируется со словом «Field» - поле в БД.

Для рассмотренного выше случая обновления количества продукта, уже существующего в корзине, при добавлении из каталога, получим следующие запросы:

Обычный вариант:

```
UPDATE "basketapp_basket"
SET
  "user_id" = 1, "product_id" = 15, "quantity" = 3,
  "add_datetime" = '2018-01-29 15:20:39.308900'
WHERE "basketapp_basket"."id" = 144
```

С использованием F-объекта:

```
UPDATE "basketapp_basket"
SET
  "user_id" = 1, "product_id" = 15,
  "quantity" = ("basketapp_basket"."quantity" + 1),
  "add_datetime" = '2018-01-29 15:20:39.308900'
WHERE "basketapp_basket"."id" = 144
```

Аналогичные правки делаем в функциях обновления остатков продуктов «product_quantity_update_save()» и «product_quantity_update_delete()» в приложении «ordersapp».

Реализация такого подхода позволит повысить производительность проекта. Так мы исключаем преобразования, происходящие между разными слоями, которые всегда имеют высокую вычислительную стоимость. Работа на уровне БД всегда быстрее, чем на уровне python.

Внимание: если значение атрибута еще не задано, например, при добавлении нового товара в корзину, использовать F-объект нельзя. Применяем его только для обновления существующих значений.

Если в дальнейшем вам понадобится значение обновленного атрибута, необходимо получить его при помощи метода «.refresh_from_db()».

Объект «F» - задаем скидку на товары в категории за один запрос

При обновлении значения атрибута нескольких объектов сразу, особенно эффективно сочетание F-объекта и метода «.update()». Вместо двух запросов (получения и сохранения значения) получим один.

Например, реализуем применение скидки к товарам выбранной категории в админке. Добавим [целочисленное поле](#) к форме:

geekshop/adminapp/forms.py

```

...
class ProductCategoryEditForm(forms.ModelForm):
    discount = forms.IntegerField(label='скидка', required=False, \
                                  min_value=0, max_value=90, initial=0)

    class Meta:
        model = ProductCategory
        # fields = '__all__'
        exclude = ()
...

```

Чтобы избежать проблем с валидацией мы задали аргумент «required=False», начальное значение установили через аргумент «initial=0». Также, определили диапазон допускаемых значений. Вместо атрибута разрешенных полей «fields» задаем «exclude» - кортеж с исключенными из отображения полями. Следующий шаг - контроллер:

geekshop/adminapp/views.py

```

...
class ProductCategoryUpdateView(UpdateView):
    model = ProductCategory
    template_name = 'adminapp/category_update.html'
    success_url = reverse_lazy('admin:categories')
    form_class = ProductCategoryEditForm

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = 'категории/редактирование'
        return context

    def form_valid(self, form):
        if 'discount' in form.cleaned_data:
            discount = form.cleaned_data['discount']
            if discount:
                self.object.product_set.\
                    update(price=F('price') * (1 - discount / 100))
                db_profile_by_type(self.__class__, 'UPDATE', \
                                   connection.queries)

        return super().form_valid(form)
...

```

Чтобы на форме редактирования каталога отображалось поле со скидкой, вместо атрибута «fields» задаем в классе «ProductCategoryUpdateView» атрибут «form_class». Для выполнения дополнительных действий при сохранении категории переопределяем метод «.form_valid()». Если данные о скидке есть и она не равна нулю - выполняем запрос. Для профилирования добавили функцию «db_profile_by_type()»:

```
UPDATE "mainapp_product"
SET "price" = CAST(("mainapp_product"."price" * 0.95) AS NUMERIC)
WHERE "mainapp_product"."category_id" = 4
```

Видим, что и в этот раз благодаря применению F-объекта Django отлично справился с задачей.

*Рекомендуем почитать материал о [предотвращении гонок при помощи F-объекта](#).

Объект «Q» - логика в запросах

До настоящего времени мы могли в запросах использовать только логическое «И» (AND). Для запросов, использующих логическое «ИЛИ» (OR) или отрицание «НЕ» (NOT), необходимо использовать [Q-объект](#) Django-ORM. Например, мы хотим вывести все продукты из категорий «Офис» и «Модерн»:

```
Product.objects.filter(Q(category__name='офис') | Q(category__name='модерн'))
```

Один Q-объект представляет одно условие поиска. Мы можем работать с ними при помощи операторов: «&», «|», «~». Для управления порядком выполнения можем использовать скобки. В принципе, можно даже комбинировать обычные условия поиска с Q-объектами, но важен порядок: первыми должны быть Q-объекты.

Для лучшего понимания работы БД создадим в папке «mainapp/management/commands» еще один служебный скрипт:

geekshop/mainapp/management/commands/learn_db.py

```
from django.core.management.base import BaseCommand
from mainapp.models import ProductCategory, Product
from django.db import connection
from django.db.models import Q
from adminapp.views import db_profile_by_type

class Command(BaseCommand):
    def handle(self, *args, **options):
        test_products = Product.objects.filter(
            Q(category__name='офис') |
            Q(category__name='модерн')
        )

        print(len(test_products))
        # print(test_products)

        db_profile_by_type('learn db', '', connection.queries)
```

Запустив его в консоли:

```
python manage.py learn_db
```

Получим запрос:

```
SELECT
    "mainapp_product"."id", "mainapp_product"."category_id",
    "mainapp_product"."name", "mainapp_product"."image",
    "mainapp_product"."short_desc", "mainapp_product"."description",
    "mainapp_product"."price", "mainapp_product"."quantity",
    "mainapp_product"."is_active"
FROM "mainapp_product"
INNER JOIN "mainapp_productcategory"
ON ("mainapp_product"."category_id" = "mainapp_productcategory"."id")
WHERE
    ("mainapp_productcategory"."name" = 'офис' OR
     "mainapp_productcategory"."name" = 'модерн')
```

Давайте теперь проверим свое понимание работы Django-ORM - сколько запросов получим, для закомментированной ранее команды:

```
print(test_products)
```

Правильный ответ: по запросу на каждый продукт плюс один. Почему? Из-за нашей реализации метода «`__str__()`» модели Product:

```
return f"{self.name} ({self.category.name})"
```

Второй вопрос: как уменьшить число запросов? Правильный ответ: использовать метод «`.select_related()`» - получим всего один запрос. Также в качестве альтернативы можете попробовать убрать вывод имени категории в методе «`__str__()`» модели Product. Тоже получите один запрос.

«Conditional-expressions» - логика на уровне QuerySet

Предположим, что в нашем магазине объявлено несколько акций:

- при оплате заказа в течение 12 часов - скидка 30%;
- при оплате заказа в течение суток - скидка 15%;
- на остальные товары - скидка 5%.

Выведем данные о величине скидки на каждый элемент заказа с учетом акции следующим образом. Сначала выводим позиции, которые попали под первую акцию в порядке увеличения выгоды. Затем позиции, которые попали под вторую акцию в порядке уменьшения выгоды. Потом - остальные позиции, снова в порядке увеличения выгоды. Чередование можно реализовать и для большего количества предложений. Таким образом получим своего рода волны, по которым можно найти самые выгодные товары на границах акций.

Чтобы сэкономить время и не создавать новых атрибутов модели, будем считать датой оплаты заказа значение его атрибута «updated». На самом деле в будущем для хранения даты оплаты можем создать, например, атрибут «paid».

Закомментируем предыдущий код и допишем новый в файл:

geekshop/mainapp/management/commands/learn_db.py

```
...
from django.db.models import F, When, Case, DecimalField, IntegerField
from datetime import timedelta
...
ACTION_1 = 1
ACTION_2 = 2
ACTION_EXPIRED = 3

action_1__time_delta = timedelta(hours=12)
action_2__time_delta = timedelta(days=1)

action_1__discount = 0.3
action_2__discount = 0.15
action_expired__discount = 0.05

action_1__condition = Q(order__updated__lte=F('order__created') +\
                        action_1__time_delta)

action_2__condition = Q(order__updated__gt=F('order__created') +\
                        action_1__time_delta) &\
                        Q(order__updated__lte=F('order__created') +\
                        action_2__time_delta)

action_expired__condition = Q(order__updated__gt=F('order__created') +\
                        action_2__time_delta)

action_1__order = When(action_1__condition, then=ACTION_1)
action_2__order = When(action_2__condition, then=ACTION_2)
action_expired__order = When(action_expired__condition, then=ACTION_EXPIRED)

action_1__price = When(action_1__condition,
                        then=F('product__price') * F('quantity') * action_1__discount)

action_2__price = When(action_2__condition,
                        then=F('product__price') * F('quantity') * -action_2__discount)

action_expired__price = When(action_expired__condition,
                              then=F('product__price') * F('quantity') * action_expired__discount)

test_orderss = OrderItem.objects.annotate(
    action_order=Case(
        action_1__order,
        action_2__order,
        action_expired__order,
        output_field=IntegerField(),
```

```

    ).annotate(
        total_price=Case(
            action_1__price,
            action_2__price,
            action_expired__price,
            output_field=DecimalField(),
        ).order_by('action_order', 'total_price').select_related()

for orderitem in test_orderss:
    print(f'{orderitem.action_order:2}: заказ №{orderitem.pk:3}: \
        {orderitem.product.name:15}: скидка \
        {abs(orderitem.total_price):6.2f} руб. | \
        {orderitem.order.updated - orderitem.order.created}')

```

Сначала задаем константы для сортировки результатов и необходимые параметры, среди которых:

- **<имя акции>__time_delta** - время действия акции;
- **<имя акции>__discount** - скидка по акции.

Далее выполняем цепочку методов в менеджере модели элемента заказа OrderItem. При помощи метода «.annotate()» добавляем поля аннотаций «action_order» и «total_price» к каждому объекту QuerySet. Сортируем результаты по этим полям и подгружаем данные связанных моделей для уменьшения количества запросов.

Для заполнения полей аннотаций используем объект Django-ORM класса «[Case](#)», который позволяет реализовать в запросах логику условного оператора «if», «then», «elif», «then».

В конструктор «Case» позиционными аргументами передаем заранее созданные объекты классе «[When](#)», возвращающие данные при выполнении определенного условия. Если необходимо, можно прописать конструкторы «When» прямо в выражениях «Case». Тип выводимого значения задается в конструкторе «Case» именованным аргументом «output_field».

Первый аргумент для конструктора «When» - условие, второй - возвращаемое значение. В нашем случае это либо константа для сортировки («ACTION_1», «ACTION_2» или «ACTION_EXPIRED»), либо - цена с учетом скидки. Для удобства восприятия и дальнейшего сопровождения кода поместили условия в отдельные переменные «action_1__condition», «action_2__condition» и «action_expired__condition».

Полную стоимость заказа вычисляем на уровне БД при помощи уже знакомых F-объектов. Для сортировки по убыванию просто меняем знак, умножив на «action_2__discount».

При выводе результатов используем F-строки Python 3.6:

1: заказ № 33: прогресс 2	: скидка 9347.20 руб.	0:00:00.292017
1: заказ № 72: прогресс 2	: скидка 9347.20 руб.	3:28:09.274000
1: заказ № 49: премиум 2	: скидка 9633.56 руб.	1:11:02.180239
1: заказ № 10: люкс	: скидка 9789.59 руб.	0:00:00.120007
1: заказ № 70: комфорт 2	: скидка 11061.60 руб.	3:28:09.274000
1: заказ № 12: рим 2	: скидка 12001.34 руб.	6:50:43.975887
1: заказ № 90: люкс 3	: скидка 12189.59 руб.	0:01:24.988500
1: заказ № 78: прогресс 3	: скидка 23494.39 руб.	0:05:39.098500
2: заказ № 7: прогресс 3	: скидка 5873.60 руб.	19:47:51.618386
2: заказ № 4: люкс 3	: скидка 3047.40 руб.	19:47:51.618386
2: заказ № 8: венеция	: скидка 2659.59 руб.	19:35:31.366167
2: заказ № 2: тоскана	: скидка 1833.30 руб.	19:47:51.618386
2: заказ № 15: прогресс 2	: скидка 1168.40 руб.	19:35:31.366167
2: заказ № 1: стандарт	: скидка 1137.15 руб.	19:47:51.618386
3: заказ № 23: стандарт	: скидка 94.76 руб.	14 days, 16:59:12.588632
3: заказ № 43: комфорт 1	: скидка 149.48 руб.	10 days, 7:45:01.028625
3: заказ № 44: премиум	: скидка 179.37 руб.	10 days, 7:45:01.028625
3: заказ № 25: прогресс	: скидка 339.47 руб.	14 days, 16:59:12.588632
3: заказ № 45: прогресс 2	: скидка 389.47 руб.	10 days, 7:45:01.028625
3: заказ № 24: прогресс 3	: скидка 489.47 руб.	14 days, 16:59:12.588632
3: заказ № 98: прогресс 3	: скидка 489.47 руб.	13 days, 1:51:18.604400
3: заказ № 22: люкс 3	: скидка 507.90 руб.	14 days, 16:59:12.588632
3: заказ № 46: премиум	: скидка 538.11 руб.	11 days, 23:04:04.676603
3: заказ № 48: венеция	: скидка 709.22 руб.	11 days, 23:04:04.676603
3: заказ № 47: новинка	: скидка 804.22 руб.	11 days, 23:04:04.676603
3: заказ № 99: тоскана	: скидка 916.65 руб.	13 days, 1:51:18.604400
3: заказ № 26: тоскана	: скидка 1222.20 руб.	14 days, 16:59:12.588632
3: заказ № 58: люкс 3	: скидка 2031.60 руб.	10 days, 4:50:16.478147

Видим, что в данном случае по «акции 1» выгоднее всего было приобретение товара «прогресс 3» в заказе №78, оплаченного в течение 5 минут 39 секунд. По «акции 2» - удачнее всего был куплен товар «прогресс 3» в заказе №7, оплаченный в течение 19 часов 47 минут. Среди остальных товаров наибольшая выгода получилась для позиции «люкс 3» в заказе №58, оплаченном более 10 дней назад.

Этим, достаточно сложным примером мы заканчиваем работу с Django-ORM в курсе. Обязательно поэкспериментируйте с ним - можно попробовать вычислять срок оплаты товара прямо в запросе и добавлять при помощи метода «.annotate()» как поле, можно добавить фильтрацию по конкретному пользователю и т.д.

Попробуйте самостоятельно реализовать агрегацию данных при помощи метода «[.aggregate\(\)](#)».

Внимание: одно из ограничений базы данных «sqlite» - мы не можем получить разницу двух дат в запросе (всегда будет возвращаться нулевая разница). При работе с другими БД все нормально.

Тестирование в Django

Еще с первых уроков мы заметили в папках приложений файлы с именем «tests.py» - именно в них и будем писать тесты. Когда их станет много - можем создавать новые файлы с префиксом «tests_».

Основа тестов Django - класс «[django.test.TestCase](#)», реализованный на базе классов Python-модуля «[unittest](#)».

Тестирование работоспособности

Обычно тестирование начинают с проверки работоспособности - «Smoke test». На 6 уроке при помощи утилиты «siege» мы уже выполняли такое тестирование перед нагрузочным. Рассмотрим его реализацию на уровне Django для приложения «mainapp». Экспортируем накопившиеся в базе данные, чтобы их можно было использовать в тестировании:

```
python manage.py dumpdata -e=contenttypes -e=auth -o test_db.json
```

Пишем тест:

geekshop/mainapp/tests.py

```
from django.test import TestCase
from django.test.client import Client
from mainapp.models import Product, ProductCategory
from django.core.management import call_command

class TestMainappSmoke(TestCase):
    def setUp(self):
        call_command('flush', '--noinput')
        call_command('loaddata', 'test_db.json')
        self.client = Client()

    def test_mainapp_urls(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

        response = self.client.get('/contact/')
        self.assertEqual(response.status_code, 200)

        response = self.client.get('/products/')
        self.assertEqual(response.status_code, 200)

        response = self.client.get('/products/category/0/')
        self.assertEqual(response.status_code, 200)

        for category in ProductCategory.objects.all():
            response = self.client.get(f'/products/category/{category.pk}/')
            self.assertEqual(response.status_code, 200)

        for product in Product.objects.all():
            response = self.client.get(f'/products/product/{product.pk}/')
            self.assertEqual(response.status_code, 200)

    def tearDown(self):
        call_command('sqlsequencereset', 'mainapp', 'authapp', 'ordersapp', \
                    'basketapp')
```

Все тесты создаются в виде классов-потомков `TestCase`. Прописываем код подготовки к тестам в методе `«.setUp()»`: создаем объект класса `«Client»` для отправки запросов, очищаем базу и импортируем данные при помощи функции `«call_command('loaddata', 'test_db.json')»`, имитирующей выполнение команды в терминале:

```
python manage.py loaddata test_db.json
```

Зачем импортировать данные? В ходе тестирования Django не работает с настоящей базой проекта. Перед тестами создается чистая виртуальная база. Иногда достаточно создать несколько объектов вручную прямо в методе `«.setUp()»`, но в данном случае для формирования URL адресов нужны реальные данные о категориях и продуктах.

Так как разные базы данных по-разному работают с индексами при создании новых элементов - добавили в метод `«.tearDown()»`, выполняющийся всегда по завершении тестов в классе, команду сброса индексов:

```
call_command('sqlsequencereset', 'mainapp', 'authapp', 'ordersapp', 'basketapp')
```

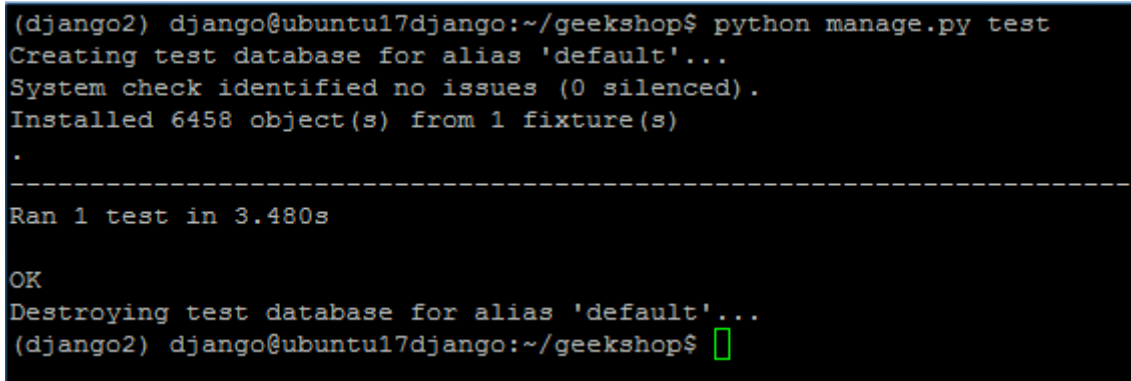
Если убрать эту команду при тестировании с использованием БД PostgreSQL возникнут проблемы. Вообще, если у вас будут затруднения с импортом данных в ходе тестирования - откажитесь от него и создавайте объекты вручную.

Запрос серверу можем посылать при помощи методов `«.get()»` и `«.post()»` объекта `«Client»`. Объект ответа сохраняем в переменной `«response»`. Рекомендуем изучить его атрибуты и методы при помощи python-функции `«dir()»`. В данном тесте делаем *только* классическую проверку по значению атрибута `«status_code»` - [код ответа сервера](#): если все хорошо, то должны получить `«200»`.

Для проверки соответствия реального и ожидаемого значения в тестах используем метод `«.assertEqual()»` класса `TestCase`.

Обязательно выключаем режим отладки в файле настроек проекта перед запуском тестирования:

```
python manage.py test
```



```
(django2) django@ubuntu17django:~/geekshop$ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
Installed 6458 object(s) from 1 fixture(s)
.
-----
Ran 1 test in 3.480s

OK
Destroying test database for alias 'default'...
(django2) django@ubuntu17django:~/geekshop$
```

Тест успешно пройден. Видим, что действительно база была создана и уничтожена. Вопрос - откуда могло появиться такое большое количество объектов? Ответ - это данные о сессиях, которые накопились во время нагрузочного тестирования. Исключим эти данные из экспорта:

```
python manage.py dumpdata -e=contenttypes -e=auth -e=sessions -o test_db.json
```

Внимание: если при запуске теста в *nix-системе вы увидели ошибку:

```
(django2) django@ubuntu17django:~/geekshop$ python manage.py test
Creating test database for alias 'default'...
Got an error creating the test database: permission denied to create database
```

Необходимо разрешить нашей учетной записи django создавать новые базы на сервере PostgreSQL:

```
sudo -u postgres psql
ALTER USER django CREATEDB;
```

Тестирование аутентификации пользователя

Пишем тесты в приложении «authapp»:

geekshop/authapp/tests.py

```
from django.test import TestCase
from django.test.client import Client
from authapp.models import ShopUser
from django.core.management import call_command

class TestUserManagement(TestCase):
    def setUp(self):
        call_command('flush', '--noinput')
        call_command('loaddata', 'test_db.json')
        self.client = Client()

        self.superuser = ShopUser.objects.create_superuser('django2', \
            'django2@geekshop.local', 'geekbrains')

        self.user = ShopUser.objects.create_user('tarantino', \
            'tarantino@geekshop.local', 'geekbrains')

        self.user_with__first_name = ShopUser.objects.create_user('umaturman', \
            'umaturman@geekshop.local', 'geekbrains', first_name='Ума')

    def test_user_login(self):
        # главная без логина
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)
        self.assertTrue(response.context['user'].is_anonymous)
        self.assertEqual(response.context['title'], 'главная')
        self.assertNotContains(response, 'Пользователь', status_code=200)
        # self.assertNotIn('Пользователь', response.content.decode())

        # данные пользователя
```

```

self.client.login(username='tarantino', password='geekbrains')

# ЛОГИНИМСЯ
response = self.client.get('/auth/login/')
self.assertFalse(response.context['user'].is_anonymous)
self.assertEqual(response.context['user'], self.user)

# Главная после логина
response = self.client.get('/')
self.assertContains(response, 'Пользователь', status_code=200)
self.assertEqual(response.context['user'], self.user)
# self.assertIn('Пользователь', response.content.decode())

def tearDown(self):
    call_command('sqlsequencereset', 'mainapp', 'authapp', 'ordersapp', \
                'basketapp')

```

Здесь мы для иллюстрации создали новых пользователей в «.setUp()». Сначала пробуем зайти на главную страницу без логина. Благодаря работе контекстного процессора «django.contrib.auth.context_processors.auth» объект пользователя всегда есть в контексте. Получаем его по ключу и проверяем при помощи метода «.assertTrue()» атрибут анонимности:

```
assertTrue(response.context['user'].is_anonymous)
```

Также можем проверить значения переменных, которые передаются из контроллера, например, заголовок страницы «title».

Если мы не залогинились - в меню не должно быть личного кабинета. Проверку можно выполнить по-разному:

- при помощи метода «.assertNotContains()» - проверяет, что в ответе нет заданного текста, и код ответа;
- при помощи метода «.assertNotIn()» - аналог Python-кода «not in», но передавать нужно уже декодированный контент «response.content.decode()».

Второй способ получается более универсальным. Остальная часть кода должна быть очевидна. При запуске тестов все должно быть хорошо. Если уберем импорт данных из базы - получим ошибку:

```

hot_product = get_hot_product()
File "/home/django/geekshop/mainapp/views.py", line 101, in get_hot_product
    return random.sample(list(products), 1)[0]
File "/usr/lib/python3.6/random.py", line 317, in sample
    raise ValueError("Sample larger than population or is negative")
ValueError: Sample larger than population or is negative

-----
Ran 2 tests in 1.099s

FAILED (errors=1)
Destroying test database for alias 'default'...
(django2) django@ubuntu17django:~/geekshop$

```

Причем она может проявиться не сразу - пока данные о продуктах в каталоге хранятся в кеше ее не будет. Причина ошибки - строка в функции «get_hot_product()» в приложении «mainapp»:

```
return random.sample(list(products), 1)[0]
```

Если база данных пуста, то список продуктов тоже пустой и, как следствие, мы получаем ошибку. В реальном проекте такая ситуация маловероятна, но для надежности лучше поработать с этой ошибкой, ведь ее исправление затронет много кода. Пока же вернем импорт данных из базы, чтобы тестировать реальную работу сайта.

Добавим тест переадресации при доступе к корзине:

geekshop/authapp/tests.py

```
...
class TestUserManagement(TestCase):
    ...
    def test_basket_login_redirect(self):
        # без логина должен переадресовать
        response = self.client.get('/basket/')
        self.assertEqual(response.url, '/auth/login/?next=/basket/')
        self.assertEqual(response.status_code, 302)

        # с логином все должно быть хорошо
        self.client.login(username='tarantino', password='geekbrains')

        response = self.client.get('/basket/')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(list(response.context['basket']), [])
        self.assertEqual(response.request['PATH_INFO'], '/basket/')
        self.assertIn('Ваша корзина, Пользователь', response.content.decode())
    ...
```

Через значение атрибута «.url» объекта ответа response проверили правильность переадресации. Также проверили значение ключа «PATH_INFO» в объекте запроса «request».

Проверяем выход из системы:

geekshop/authapp/tests.py

```
...
...
def test_user_logout(self):
    # данные пользователя
    self.client.login(username='tarantino', password='geekbrains')

    # логинимся
    response = self.client.get('/auth/login/')
    self.assertEqual(response.status_code, 200)
    self.assertFalse(response.context['user'].is_anonymous)
```



```

# ВЫХОДИМ ИЗ СИСТЕМЫ
response = self.client.get('/auth/logout/')
self.assertEqual(response.status_code, 302)

# ГЛАВНАЯ ПОСЛЕ ВЫХОДА
response = self.client.get('/')
self.assertEqual(response.status_code, 200)
self.assertTrue(response.context['user'].is_anonymous)

...

```

Последняя и самая интересная проверка в этом приложении - регистрация пользователя с отправкой подтверждения по почте. *Напомним:* для отправки почты в файле настроек обязательно должна быть константа имени домена «DOMAIN_NAME»:

geekshop/authapp/tests.py

```

...
from django.conf import settings
...
...
def test_user_register(self):
    # ЛОГИН БЕЗ ДАННЫХ ПОЛЬЗОВАТЕЛЯ
    response = self.client.get('/auth/register/')
    self.assertEqual(response.status_code, 200)
    self.assertEqual(response.context['title'], 'регистрация')
    self.assertTrue(response.context['user'].is_anonymous)

    new_user_data = {
        'username': 'samuel',
        'first_name': 'Сэмюэл',
        'last_name': 'Джексон',
        'password1': 'geekbrains',
        'password2': 'geekbrains',
        'email': 'sumuel@geekshop.local',
        'age': '21'}

    response = self.client.post('/auth/register/', data=new_user_data)
    self.assertEqual(response.status_code, 302)

    new_user = ShopUser.objects.get(username=new_user_data['username'])

    activation_url =
f"{settings.DOMAIN_NAME}/auth/verify/{new_user_data['email']}/{new_user.activation_key}/"

    response = self.client.get(activation_url)
    self.assertEqual(response.status_code, 200)

    # ДАННЫЕ НОВОГО ПОЛЬЗОВАТЕЛЯ
    self.client.login(username=new_user_data['username'], \
                      password=new_user_data['password1'])

```

```

# ЛОГИНИМСЯ
response = self.client.get('/auth/login/')
self.assertEqual(response.status_code, 200)
self.assertFalse(response.context['user'].is_anonymous)

# проверяем главную страницу
response = self.client.get('/')
self.assertContains(response, text=new_user_data['first_name'], \
                    status_code=200)

def test_user_wrong_register(self):
    new_user_data = {
        'username': 'teen',
        'first_name': 'Мэри',
        'last_name': 'Поппинс',
        'password1': 'geekbrains',
        'password2': 'geekbrains',
        'email': 'merypoppins@geekshop.local',
        'age': '17'}

    response = self.client.post('/auth/register/', data=new_user_data)
    self.assertEqual(response.status_code, 200)
    self.assertFormError(response, 'register_form', 'age', \
                        'Вы слишком молоды!')

...

```

Проверили правильную и неправильную по возрасту регистрацию. Передачу данных регистрируемого пользователя с формы реализуем при помощи метода «.post()», в который передаем их по ключу «data». Самая сложная часть теста - сформировать адрес подтверждения «activation_url». После подтверждения необходимо залогиниться и проверить ссылку на личный кабинет в меню на главной странице - в ней должно быть имя пользователя.

При неправильной регистрации - ввели некорректный возраст и проверили ошибку формы при помощи метода «.assertFormError()»: второй аргумент - имя формы, третий - поле. В четвертом - передаем текст ожидаемой ошибки.

Обращаем ваше внимание, что при неправильной регистрации код сервера «200», а не «302», как в случае переадресации при корректной регистрации.

Тестирование методов моделей

Еще один важный вид тестирования - проверка работы методов моделей. Ведь именно в них реализована большая часть логики проекта. Рассмотрим подход к этому вопросу на примере моделей продуктов и категорий продуктов:

geekshop/mainapp/tests_products.py

```

from django.test import TestCase
from mainapp.models import Product, ProductCategory

class ProductsTestCase(TestCase):
    def setUp(self):
        category = ProductCategory.objects.create(name="стулья")
        self.product_1 = Product.objects.create(name="стул 1",
                                                category=category,
                                                price=1999.5,
                                                quantity=150)

        self.product_2 = Product.objects.create(name="стул 2",
                                                category=category,
                                                price=2998.1,
                                                quantity=125,
                                                is_active=False)

        self.product_3 = Product.objects.create(name="стул 3",
                                                category=category,
                                                price=998.1,
                                                quantity=115)

    def test_product_get(self):
        product_1 = Product.objects.get(name="стул 1")
        product_2 = Product.objects.get(name="стул 2")
        self.assertEqual(product_1, self.product_1)
        self.assertEqual(product_2, self.product_2)

    def test_product_print(self):
        product_1 = Product.objects.get(name="стул 1")
        product_2 = Product.objects.get(name="стул 2")
        self.assertEqual(str(product_1), 'стул 1 (стулья)')
        self.assertEqual(str(product_2), 'стул 2 (стулья)')

    def test_product_get_items(self):
        product_1 = Product.objects.get(name="стул 1")
        product_3 = Product.objects.get(name="стул 3")
        products = product_1.get_items()

        self.assertEqual(list(products), [product_1, product_3])

```

Код разместили в отдельном файле - так удобнее при большом количестве тестов. Создали все объекты вручную - обошлись без импорта данных.

Проверили работу метода «.get()» менеджера модели и корректность преобразования объектов в строку. А также метод «.get_items()», позволяющий получить остальные продукты в той же категории, что у текущего продукта. Сред них не должно быть неактивного продукта «product_2». В нашем случае все правильно.

Итак, теперь при запуске должно выполняться корректно 9 тестов. Можно тестировать и конкретное приложение:

```
python manage.py test authapp
```

Внимание: не забываем включить защиту CSRF в файле настроек проекта!

На этом мы завершаем курс. Всем спасибо за работу!

Практическое задание

1. Исправить баг при управлении активностью категории. Активность товаров тоже должна обновляться.
2. Реализовать возможность сделать скидку на товары категории в админке при помощи метода `«.update()»`.
3. Поработать с F-объектом, убедиться, что обновление значений выполняется на уровне БД, а не в python-коде.
4. Написать несколько запросов с логическим «ИЛИ». Написать сложный запрос в базу с использованием «conditional-expressions».
5. Провести тестирование работоспособности одного из приложений.
6. Протестировать процесс логина пользователя и переадресацию при доступе корзине.
7. Исправить ошибку в функции `«get_hot_product()»`.
8. Написать тесты для методов моделей проекта.
9. В качестве защиты курсового проекта необходимо записать в любой удобной для вас программе видеоролик (скринкаст) продолжительностью 1-5 минут. Представьте, что вам необходимо презентовать вашу работу заказчику или аудитории. В скринкасте расскажите о вашем проекте, продемонстрируйте его возможности и функционал. Ссылку на видео приложите к практическому заданию, например, в комментарии к уроку. И не забудьте открыть доступ на просмотр! :) Видеопрезентация продукта развивает у вас дополнительные мягкие навыки и является обязательной для засчитывания курсового проекта.

Дополнительные материалы

Все то, о чем сказано в методичке, но подробнее:

1. [Метод «.update\(\)»](#)
2. [F-выражения в Django](#)
3. [Предотвращение гонок при помощи F-объекта](#)
4. [Объект «Q»](#)
5. [«Conditional-expressions» в Django](#)
6. [Метод «.annotate\(\)»](#)
7. [Класс «When»](#)
8. [Класс «Case»](#)
9. [Агрегация при помощи «.aggregate\(\)»](#)

10. [Тестирование на «дым» \(Smoke test\)](#)
11. [Коды состояний HTTP](#)
12. [Утверждения \(assertions\) в тестах](#)
13. [Написание тестов](#)
14. [Запуск тестов](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. [Официальная документация](#)