Урок 3

Модели + ORM = данные

Краткая теория баз данных. Введение в django-ORM. Подключение и создание базы данных. Несколько слов о миграциях. Работа с данными. Встроенная админка. Пространства имен.

Введение в Django ORM

Django поддерживает четыре системы управления базами данных:

- PostgreSQL;
- SQLite 3;
- MySQL;
- Oracle.

Выбор БД — важное решение. Для небольших проектов (как наш учебный) подойдет файловая БД SQLite 3. Преимущества: не нужно разворачивать и настраивать сервер БД. Для реальных проектов чаще всего используют серверные БД PostgreSQL и MySQL. Они хорошо масштабируются и бесплатны. При необходимости можно использовать «тяжелую артиллерию» в виде Oracle, но надо покупать лицензию.

Django как фреймворк при работе с БД дает возможность работать с таблицами как с объектами. Эта технология называется ORM (Object-Relational Mapping, объектно-реляционное отображение). По сути это абстракция над реальным движком БД и языком SQL. Преимущество: мы можем «забыть», какая СУБД в проекте, и просто сосредоточиться на логике работы с данными.

Как и в случае с шаблонами, за работу с БД отвечает соответствующий backend, который прописывается в константе **DATABASES** файла **settings.py**:

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

Просто изменив настройку и выполнив миграции, можно перейти на другую СУБД! Это то, за что мы любим фреймворки.

Сразу поясним, что **миграции** — это процесс преобразования модели данных (файл **models.py**) в реальные таблицы БД.

Замечание:

> Если меняете атрибуты модели — нужно **всегда** выполнять миграции, чтобы скорректировать структуру таблиц БД. При изменении методов работы с данными в модели миграции делать **не нужно**!

Первая модель в проекте

Модели отображают информацию о данных, с которыми вы работаете. Они содержат поля и методы для обработки данных. Обычно одна модель представляет одну таблицу в базе данных. Создадим модель категории товара в магазине.

mainapp/models.py

```
from django.db import models

class ProductCategory(models.Model):
    name = models.CharField(verbose_name='имя', max_length=64, unique=True)
    description = models.TextField(verbose_name='описание', blank=True)

    def __str__(self):
        return self.name
```

С точки зрения Python модель — это класс, унаследованный от Django-класса **models.Model**. Имя класса преобразуется в имя таблицы БД вида <имя приложения>_<имя таблицы>: **mainapp_productcategory**.

Атрибуты класса **name** и **description** становятся именами полей таблицы. При этом всегда автоматически создается поле с первичным ключом **id**.

Когда выбираем функцию, создающую атрибут модели, необходимо мыслить как при выборе типа поля в таблице БД (на примере MySQL):

- **VARCHAR -> models.CharField**
- **TEXT -> models.TextField**
- **FLOAT -> models.FloatField**
- **INT -> models.IntegerField**
- **INT UNSIGNED-> models.PositiveIntegerField**

Более подробно — в документации Django.

При создании таблицы в БД обычным способом (например, **phpMyAdmin**) для ее полей задаем дополнительные параметры: длину поля, начальное значение, наличие знака, уникальность. В Django ORM это делается при помощи аргументов. Их набор зависит от типа атрибута. Например, для **models.CharField** обязательно задаем максимальную длину поля **max_length**. Остальные аргументы — необязательные. Если поле должно быть уникальным — **unique=True**; псевдоним (подробное имя) — **verbose_name**. Значение поля по умолчанию задается аргументом **default** (например, default=0).

Замечание:

> При работе с **необязательными** для ввода полями в Django есть особенность: для символьных полей используем аргумент **blank=True**, а для числовых — либо аргумент **null=True**, либо просто задается начальное значение. Будьте внимательны!

Есть типы полей, требующие более пристального внимания:

- DateTimeField — используется для записи даты и времени. Рассмотрим параметры:
    - **auto_now**: при **каждом** сохранении объекта проставляется текущее время (полезно для сохранения времени последнего обновления);
    - **auto_now_add:** при **первом** сохранении объекта проставляется текущее время (полезно для сохранения времени создания объекта).

- DecimalField — удобно для хранения финансовых величин. Стоит обратить внимание на параметры:
    - **max_digits** — максимальное количество цифр;
    - **decimal_places** — количество знаков после запятой.

Когда модель создана, необходимо выполнить миграции. Для этого запускаем интерпретатор командной строки в корне проекта и выполняем команду:

```
python manage.py migrate
```

В результате Django создает в БД структуру таблиц: таблицы пользователей, групп пользователей, сессий, типов контента и другие. Вы можете открыть файл **db.sqlite3** в корне проекта и убедиться в этом — например, при помощи бесплатной программы DB Browser for SQLite. Таблицы нашей модели пока нет. Чтобы она появилась, сначала создадим файл миграций:

```
python manage.py makemigrations
```

Если модель не содержит ошибок, в папке '**mainapp/migrations**' появится файл '**0001_initial.py**' — это и есть первая (initial) миграция. Обязательно посмотрите его содержимое — по сути это трансляция кода модели в python-код, который создаст таблицу в БД. Но **ни в коем случае не изменяйте** содержимое этого файла! Когда будете модифицировать модель и создавать новые миграции, будут появляться новые файлы миграций. Это позволяет переходить от одного состояния модели к другому (откат миграций). В нашем курсе этот вопрос не рассматривается.

Когда миграция **создана** — необходимо ее **выполнить**:

```
python manage.py migrate
```

Теперь можно проверять — в БД должна появиться таблица '**mainapp_productcategory**'. В дальнейшем при изменениях в атрибутах модели мы всегда будем создавать и выполнять миграции. Рекомендуем по аналогии с **run.bat** сделать файл **migrate.bat** с выполненным только что кодом командной строки.

Замечание:

> Если при создании миграции Django не видит изменений в моделях — проверьте, прописано ли приложение в файле **settings.py**. Можно создать миграцию для конкретного приложения, если, например, выполнить команду **python manage.py makemigrations mainapp**. Но проще, когда это происходит автоматически.

Связанные модели

Первая модель была очень простой. Теперь создадим модель товара, в которой будет больше полей. Нам предстоит решить важную задачу: связать поле «категория продукта» с ранее созданной моделью категорий. Если вспомнить теорию БД, существует три вида связей между таблицами:

- **«один к одному»** (one-to-one) — например, если мы хотим хранить описание товара в отдельной таблице;

- **«один ко многим»** (one-to-many) — используется чаще всего и позволяет, например, связать много записей в одной таблице (товары) с одной записью в другой (категория товара);

- **«многие ко многим»** (many-to-many) — в качестве примера можно рассмотреть таблицу с названиями книг и таблицу с книжными магазинами: книга может быть представлена во многих магазинах, а в магазине может продаваться много книг.

Для модели продуктов магазина нам понадобится связь «**один ко многим**» с моделью «категория продукта». Добавим в файл **mainapp/models.py** код:

mainapp/models.py

```
class Product(models.Model):
    category = models.ForeignKey(ProductCategory, on_delete=models.CASCADE)
    name = models.CharField(verbose_name='имя продукта', max_length=128)
    image = models.ImageField(upload_to='products_images', blank=True)
    short_desc = models.CharField(verbose_name='краткое описание продукта',
max_length=60, blank=True)
    description = models.TextField(verbose_name='описание продукта', blank=True)
    price = models.DecimalField(verbose_name='цена продукта', max_digits=8,
decimal_places=2, default=0)
    quantity = models.PositiveIntegerField(verbose_name='количество на складе',
default=0)

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

Разберем новые поля модели:

1. **models.ForeignKey** — это связь с моделью **ProductCategory**. Обязательный первый аргумент — имя связанной модели, второй — действие при удалении: если удалить категорию, все ее продукты тоже будут удалены.

2. **models.ImageField** — поле для хранения изображения. Аргумент **upload_to** позволяет задать имя папки (относительно корня медиафайлов в проекте), где будут храниться изображения. Второй аргумент **blank=True** задали, чтобы загрузка изображения была необязательной.

3. **models.DecimalField** — поле для хранения цены (более удобная альтернатива **models.FloatField**). Аргумент **max_digits** — это **общее** количество разрядов (включая десятичные), **decimal_places** — число десятичных разрядов.

Создаем и выполняем миграции.

Посмотрим, как реализована в Django связь моделей на уровне БД. В таблице '**mainapp_products**' создано поле **category_id** (<имя атрибута модели>_id), в котором и хранится внешний ключ (**foreign key**) категории товара.

Рекомендуем просмотреть типы остальных полей таблицы. Вы обнаружите, что **models.ImageField** превратилось в обычное строковое поле **VARCHAR**. Но модель в Django — это не только поля в таблицах, но и действия. Например, когда мы используем **models.ImageField**, Django обеспечивает загрузку и сохранение изображения (рассмотрим подробнее позже).

Замечание:

При заполнении БД будьте внимательны: всегда **сначала** создавайте запись в связанной модели (категории продуктов) и только **потом** относящиеся к ней записи (продукты)!

Настройка проекта для работы с медиафайлами

Мы уже умеем работать со статическими файлами в Django. Но в любом реальном проекте пользователи или администраторы будут загружать медиафайлы на сайт (фотографии, музыку, видео). Откроем файл **settings.py** и пропишем в нем код:

```
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

Похоже на работу со статическими файлами? Главное отличие: **MEDIA_ROOT** — это не кортеж, а строка. Второе — чтобы Django раздавал медиафайлы на этапе разработки, необходимо добавить в файл **urls.py** строки:

```
from django.conf import settings
from django.conf.urls.static import static

if settings.DEBUG:
   urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Смысл этого кода — сообщить Django, что нужно папку на диске **MEDIA_ROOT** сделать доступной по сетевому адресу **MEDIA_URL**.

Не забудьте создать папку **'/media/'** в корне проекта.

Работа с моделью в консоли

Базу данных в Django можно заполнять данными по-разному:

- вручную через консоль (полезно несколько раз попробовать);
- при помощи админки (встроенной в Django или собственной);
- импорт из файла с данными (json, yaml).

Работа через консоль

Научиться работать с ORM через консоль важно: так вы быстро сможете добавлять данные в базу и читать из нее, не создавая формы и контроллеры. И еще будете видеть весь процесс работы с ORM. Рано или поздно все равно придется этому научиться, так что приступим.

Создадим категорию продукта

Запускаем консоль **python** в корне проекта:

```
python manage.py shell
```

Импортируем модель категорий:

```
from mainapp.models import ProductCategory
```

Создаем объект класса **ProductCategory**:

```
new_category = ProductCategory(name='Стулья')
```

В конструкторе задаем обязательные атрибуты модели (можно задать и все атрибуты). Теперь объект необходимо сохранить:

```
new_category.save()
```

Если этого не сделать, объект так и останется только в памяти компьютера, но не запишется в таблицу. Можете посмотреть базу — должна быть первая запись в '**mainapp_productcategory**'. Убедимся, что запись появилась, через консоль:

```
categories = ProductCategory.objects.all()
categories
```

Вы должны увидеть список <**QuerySet**>, содержащий записи категорий в базе. Так как мы прописали в моделях магический метод **__str__**, этот список хорошо читается. Все действия с моделями в Django будем выполнять через встроенный **менеджер модели objects**. В дальнейшем вы можете создать свои менеджеры моделей. Метод **all()** эквивалентен SQL-запросу **SELECT * FROM mainapp_productcategory**. Результат запроса — объект **QuerySet**, к которому мы позже вернемся. Пока будем воспринимать его как обычный список в **python**. Можем получить конкретную запись как элемент списка и отредактировать ее:

```
category1 = categories[0]
category1.description = 'отличные стулья'
category1.save()
```

Таким образом мы добавили описание категории в уже существующую запись. Можно было это сделать и сразу после создания объекта. Если захотим удалить запись из БД — вызовем метод **delete()**:

```
category1.delete()
```

Создадим продукт в категории

Когда в памяти компьютера есть объект категории (пусть в переменной **category1**), можно создавать записи объектов, связанных с этой категорией (в нашем случае — продуктов):

```
from mainapp.models import Product
new_product = Product(category=category1, name='Удобный стул', price=1979.56,
quantity=198)
new_product.save()
```

Обязательно закрепите навыки: создайте несколько категорий и продуктов в них.

В Django **проверка валидности** (соответствия) данных происходит только при работе с **формами**. Поэтому в консоли нужно быть внимательнее, чтобы не записать ошибочные данные в базу.

В случае ошибки — начинаем заново

Если в ходе работы с моделями, миграциями или данными возникли ошибки (а такое на этапе обучения должно быть):

- удаляем файл с БД (**db.sqlite3**);
- удаляем файлы с миграциями (**0001_initial.py** и т.д.);
- файл **__init__.py** в папке **'mainapp/migrations**' — **не трогать!**
- создаем миграции **python manage.py makemigrations**;
- выполняем миграции **python manage.py migrate**.

Конечно, база при этом будет пустой.

Работа через админку

Иногда работать через консоль полезно, но при больших объемах данных нужно искать другие решения по наполнению сайта и управлению контентом. Ближе к концу курса мы напишем свою админку, а пока рассмотрим возможности встроенной в Django. Как видно из файла **urls.py**, она доступна по URL-адресу:

```
127.0.0.1:8000/admin/
```

Если перейти по этому адресу, появится окно для входа в систему (доступ ограничен). Чтобы продолжить, необходимо сделать два шага.

1. Создать суперпользователя через консоль:

   ```
   python manage.py createsuperuser
   ```

Далее — ввести логин (пусть это будет **django**), почту и пароль (будем всегда создавать пароль **geekbrains**).

2. Зарегистрировать модели на сайте админки.

   В файле '**mainapp/admin.py**' пишем код:

```
from django.contrib import admin
from .models import ProductCategory, Product

admin.site.register(ProductCategory)
admin.site.register(Product)
```

Теперь можно создавать категории и продукты через админку. Попробуйте добавить к продуктам изображения. Посмотрите, какие изменения произойдут в папке '**/media/**'.

Работаем с моделями в контроллерах

Получить данные из модели в контроллере очень просто: импортируем модели и выполняем действия через менеджер:

mainapp/views.py

```
from .models import ProductCategory, Product

def main(request):
    title = 'главная'

    products = Product.objects.all()[:4]

    content = {'title': title, 'products': products}
    return render(request, 'mainapp/index.html', content)
```

Хорошая новость: мы можем обращаться к полям модели (даже связанной) прямо в шаблонах.

templates/mainapp/index.html

```
...
{% for product in products %}
  <div class="block">
    <a href="#">
      <img src="/media/{{ product.image }}" alt="{{ product.short_desc }}">
      <div class="text">
        <h3>{{ product.category.name }}</h3>
        <h4>{{ product.name }}</h4>
        <p>{{ product.description }}</p>
      </div>
    </a>
  </div>
{% endfor %}
```

```
...
```

Обращаем особое внимание на запись **product.category.name**. Поясним:

- **category** — имя атрибута в модели продукта, который соответствует связанной модели категорий;
- **name** — имя атрибута связанной модели.

Теперь можем наполнить базу данными.

* Создаем диспетчер URL в приложении

По мере развития проекта контроллеров будет все больше, и число записей в **urlpatterns** будет увеличиваться. Это может привести к путанице. Django позволяет использовать пространства имен при работе с URL-адресами. Внесем изменения в файл **urls.py**:

geekshop/urls.py

```
from django.conf.urls import include

urlpatterns = [
    path('', mainapp.main, name='main'),
    path('products/', include('mainapp.urls', namespace='products')),
    path('contact/', mainapp.contact, name='contact'),
    path('admin/', admin.site.urls),
]
```

Все изменения сводятся к одной строке:

```
path('products/', include('mainapp.urls', namespace='products'))
```

При помощи функции **include()** мы подключаем еще один файл **urls.py**, который необходимо создать в папке приложения. Аргумент **namespace='products'** позволяет обращаться в шаблонах к адресам из подключаемого файла через пространство имен.

Когда пользователь переходит на вкладку «Продукты», будем отображать индексную страницу (например, горячее предложение магазина) с меню категорий продуктов. При выборе категории должны отображаться ее товары. Для этого пропишем в новом файле **urls.py**:

mainapp/urls.py

```
from django.urls import path

import mainapp.views as mainapp

app_name = 'mainapp'

urlpatterns = [
   path('', mainapp.products, name='index'),
```

```
    path('<int:pk>/', mainapp.products, name='category'),
]
```

Дальше в проекте будем создавать подобные файлы в остальных приложениях. Благодаря пространствам имен конфликта не будет, даже если мы в другом приложении снова зададим **name='index'**.

После изменений необходимо скорректировать динамические адреса в шаблонах по принципу:

```
{% url 'products' %}     ->     {% url 'products:index' %}
```

Рассмотрим, как будет работать диспетчер адресов Django. Если зайти по адресу '**127.0.0.1:8000/products/**', сработает следующая строка главного файла **urls.py**:

```
path('products/', include('mainapp.urls', namespace='products'))
```

В соответствии с выражением от запрашиваемого адреса будет отброшена часть '**products/**', и оставшаяся (в нашем случае это пустая строка '') будет передана для обработки файлу **urls.py** в папке с приложением '**mainapp**'. В этом файле на пустую строку сработает первое выражение:

```
path('', mainapp.products, name='index')
```

Что будет дальше, уже знаем.

Теперь предположим, что пользователь выбрал категорию в меню, и произошел переход, например, по адресу '**127.0.0.1:8000/products/1/**' (здесь '1' — это id категории в базе). Значит диспетчеру '**mainapp/urls.py**' будет передано значение '**1/**', и сработает уже второе выражение. Благодаря скобкам цифра '**1**' будет выделена из адреса и передана как аргумент контроллеру. Чтобы не было ошибки, добавим этот аргумент в список:

```
def products(request, pk=None):
      print(pk)
...
```

Можете проверить работу этого механизма, прописывая вручную URL-адреса. В консоли должен выводиться либо номер категории, либо **None**.

Практическое задание

1. Настроить проект для работы с медиафайлами.
2. Создать модели в проекте (обязательно должно быть поле с изображениями) и выполнить миграции.
3. Поработать с моделями в консоли.
4. Создать суперпользователя. Настроить админку и поработать в ней.

5. Организовать работу с моделями в контроллерах и шаблонах.
6. Реализовать автоматическое формирование меню категорий по данным из модели.
7. * Создать диспетчер URL в приложении. Скорректировать динамические URL-адреса в шаблонах. Поработать с имитацией переходов по категориям в адресной строке браузера.
8. * Организовать загрузку данных в базу из файла.

Дополнительные материалы

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

1. Модели Django.
2. Миграции.
3. Встроенная админка.
4. Диспетчер URL.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Видеокурс Основы баз данных.

Примечание: Видеокурс входит в программу обучения профессии Программист Python. Надеемся вы проходите курсы в рекомендуемом порядке и уже завершили данный курс. Если нет, загляните в раздел Обучение и не затягивайте с изучением баз данных.