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

Первая модель в проекте

Связанные модели

Настройка проекта для работы с медиафайлами

Работа с моделью в консоли

- Работа через консоль
  - Создание категории продукта
  - Создание продукта в категории
  - В случае ошибки
- Работа через админку

Работа с моделями в контроллерах

*Создание диспетчера URL в приложении

Домашнее задание

Дополнительные материалы

Используемая литература

Введение в Django ORM

Django поддерживает четыре системы управления базами данных:

- PostgreSQL (http://www.postgresql.org).
- SQLite 3 (http://www.sqlite.org).
- MySQL (http://www.mysql.com).
- Oracle (http://www.oracle.com).

Выбор БД для проекта – это 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 это требует покупки лицензии.

Что дает нам Django как фреймворк при работе с БД? Возможность работать с таблицами как с объектами. Эта технология называется ORM (англ. 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 загадочное слово **миграция** – это процесс преобразования модели данных (файл **`models.py`**), в реальные таблицы БД.

Замечание: если вы измените 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. Давайте создадим модель категории товара в нашем магазине.

mainapp/models.py

```
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 языка 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 мы 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 тут.

При создании таблицы в БД обычным способом (например, `phpMyAdmin`), мы 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 задаем начальное значение. Будьте внимательны!

Есть некоторые типы полей, требующие более пристального внимания:

- `DateTimeField` – используется для записи даты и времени (изучите разницу между аргументами `auto_now` и `auto_now_add`).
- `DecimalField` – удобно для хранения финансовых величин (изучите аргументы `max_digits` и `decimal_places`).

Как уже говорилось ранее, после создания модели 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 некоторую 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 содержимое файла – 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 вы 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 мы всегда будем выполнять два этих действия: создание и выполнение миграции, при изменениях в атрибутах модели. 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 ваше приложение в файле `settings.py`. В принципе 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. Давайте теперь создадим модель товара. В ней будет больше полей. Нам предстоит решить важную задачу: связать поле «категория продукта» с ранее созданной моделью категорий. Если вспомнить теорию БД, то 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 продаваться во многих магазинах и 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='кратко', max_length=60,
blank=True)
    description = models.TextField(verbose_name='подробно', blank=True)
    price = models.DecimalField(verbose_name='цена', max_digits=8,
decimal_places=2, default=0)
    quantity = models.PositiveIntegerField(verbose_name='склад', default=0)

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

Разберем новые поля модели:

- `models.ForeignKey` – это связь с моделью `ProductCategory`: обязательный первый аргумент – имя связанной модели, второй аргумент – действие при удалении: если удалить категорию, все ее продукты тоже будут удалены;
- `models.ImageField` – поле для хранения изображения: аргумент `upload_to` позволяет задать имя папки (относительно корня медиафайлов в проекте – разберем позже), где будут храниться изображения, второй аргумент blank=True задали, чтобы загрузка изображения была необязательной;
- `models.DecimalField` – поле для хранения цены (более удобная альтернатива `models.FloatField`): аргумент `max_digits` – **общее** количество разрядов (включая десятичные), `decimal_places` – число десятичных разрядов.

Создаем и выполняем миграции.

Давайте 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 тип остальных полей таблицы. Вы, например, обнаружите, что `models.ImageFie`ld превратилось в обычное строковое поле `VARCHAR`. Однако, модель в Django – это не только поля в таблицах, но и определенные действия. Например, когда мы используем `models.ImageField` – Django обеспечит процесс загрузки и сохранения изображения (рассмотрим подробнее позже).

Замечание: при заполнении БД будьте внимательны: всегда **сначала** создаем 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. Однако в любом реальном проекте пользователи или администраторы будут загружать медиафайлы на сайт (фотографии, музыка, видео и т. д.). Давайте 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: мы сообщаем 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

Важно научиться работать с ORM через консоль. Почему? Во-первых, так вы быстро сможете добавлять данные в базу и читать из базы, без необходимости создавать формы и контроллеры. Во-вторых, вы полностью будете видеть весь процесс работы с ORM. И, в-третьих, рано или поздно вам все равно придется этому научиться. Приступим.

Создание категории 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. Благодаря тому, что 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()
```

Создание продукта в категории

Итак, когда у нас 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 свои навыки на примере создания нескольких категорий и продуктов в категориях.

Замечание: в Django **проверка валидности** (соответствия) данных происходит только при работе с **формами**, поэтому при работе в консоли нужно быть внимательными, чтобы не записать ошибочные данные в базу.

В случае ошибки

Если в ходе работы с моделями, миграциями или данными возникли ошибки (а такое на этапе обучения должно быть) – мы всегда можем начать сначала:

- 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**;
- выполняем миграции.

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 и управлению контентом сайта. Ближе к концу курса мы напишем свою админку, а пока рассмотрим некоторые возможности встроенной в Django. Как видно из файла `urls.py` она доступна по url-адресу:

`127.0.0.1:8000/admin/`

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

1. Создаем суперпользователя через консоль:

   ```
   python manage.py createsuperuser
   ```

   Далее – вводим логин (пусть это будет django), почту и пароль (будем всегда создавать пароль geekbrains).
2. Регистрируем модели на сайте админки.

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

   ```
   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/'`.

Работа 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 (и 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 ваше 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 мы можем наполнить базу данными.

*Создание диспетчера 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 = [
    url(r'^$', mainapp.main, name='main'),
    url(r'^contact/$', mainapp.contact, name='contact'),
    url(r'^products/', include('mainapp.urls', namespace='products')),

    url(r'^admin/', admin.site.urls),
]
```

Все изменения сводятся к одной строке:

```
url(r'^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 (подробнее – позже).

Пусть, мы хотим, чтобы, когда пользователь переходит на вкладку продукты – отображалась некоторая индексная страница (например, горячее предложение магазина) с меню категорий продуктов. При выборе категории должны отображаться товары этой категории. Для решения этой задачи пропишем в новом файле `urls.py`:

mainapp/urls.py

```
from django.conf.urls import url

import mainapp.views as mainapp

urlpatterns = [
    url(r'^$', mainapp.products, name='index'),
    url(r'^(\d+)/$', mainapp.products, name='category'),

]
```

Дальше в проекте мы будем создавать подобные файлы в остальных приложениях. При этом даже если мы в другом приложении снова зададим `name='index'` – благодаря пространствам имен конфликта не будет.

После наших изменений необходимо скорректировать динамические адреса в шаблонах по принципу:

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

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

```
url(r'^products/', include('mainapp.urls', namespace='products')),
```

В соответствии с регулярным выражением от запрашиваемого адреса будет отброшена часть `'products/'`, и оставшаяся часть (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 регулярное выражение:

```
url(r'^$', mainapp.products, name='index'),
```

Что будет дальше – мы 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'` – `id` категории в базе). Значит диспетчеру `'mainapp/urls.py'` будет передано значение `'1/'`, и сработает уже второе регулярное 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)
...
```

На сегодня достаточно. Можете проверить работу этого механизма, прописывая вручную URL-адреса. В консоли должен выводиться либо номер категории, либо «None».

Домашнее 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 чём 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. Курс по БД
2. Перевод документации