нетология

Работа с ORM

Максим
Филипенко

Максим Филипенко

Software Engineer at Marilyn Systems

mfiliprnko.yandex.ru | prokaktus | prokaktus

План занятия

1. База данных
2. ORM, модели
3. Запросы к БД
4. Создание БД проекта
5. Модели и View
6. Управление БД из админки Django

Базы данных

- Хранение данных;
- Обработка данных;
- Управление данными;
- Совместная работа нескольких пользователей.

Почему не файлы?

1. При хранении в файле нужно писать самостоятельно проверки структуры файла.
2. Возможна потеря данных в случае ошибок.
3. С файлом сложно работать нескольким людям.
4. Для работы с большим объемом данных нужны высокоэффективные алгоритмы:
    - Поиск нужного элемента в файле.
    - Получение связанных данных из разных файлов.
    - Сортировка данных.

СУБД — Система Управления Базами Данных

Представляют из себя: Формат хранения + код для работы с данными

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

Для разработчика: драйвер + язык запросов SQL. У каждой СУБД свой диалект SQL.

Таблица БД

Таблица (база данных) — это совокупность связанных данных, хранящихся в структурированном виде. Таблица состоит из столбцов и строк.

id	brand	name	color
1	Ford	Focus	red
2	Lada	Kalina	purple

Строка называется запись (кортеж), а столбец – поле (домен).

Поле, каждое значение которого однозначно определяет соответствующую запись, называется простым ключом или primary key (ключевым полем). В данном случае это поле `id`.

SQL

SQL — язык работы с данными:

```
-- запрашиваем разные сущности, хранящиеся в отдельных таблицах

SELECT name, birthday, gender FROM person;

SELECT brand, name, color FROM car;
```

и из связанных данных получать общую картину:

```
-- У кого какая машина
SELECT person.name, car.brand, car.name
  FROM person INNER JOIN car ON car.id = person.car_id
```

Связи между таблицами

Связь позволяет моделировать отношения между объектами. Существует 4 типа связей:

1. **Один к одному** — любому экземпляру сущности A соответствует только один экземпляр сущности B, и наоборот. Редко используется, например для расширения таблицы.
2. **Один ко многим** — любому экземпляру сущности A соответствует 0, 1 или несколько экземпляров сущности B, но любому экземпляру сущности B соответствует только один экземпляр сущности A. Например: один учитель — много учеников.

Связи между таблицами

3. **Многие к одному** — любому экземпляру сущности A соответствует только один экземпляр сущности B, но любому экземпляру сущности B соответствует 0, 1 или несколько экземпляров сущности A. Например: много учителей — один ученик (по сути в другую сторону 2.).
4. **Многие ко многим** — любому экземпляру сущности A соответствует 0, 1 или несколько экземпляров сущности B, и любому экземпляру сущности B соответствует 0, 1 или несколько экземпляров сущности A. Например: множество учеников — множество учителей.

ORM

Object-Relational Mapping

- Прослойка между базой данных и программным кодом.
- Позволяет работать с таблицами через классы в Python. Вы работаете с таблицей как с обычным классом, где колонки являются атрибутами класса.
- Позволяет опустить написание SQL-запросов и взаимодействовать с базами данных посредством языка программирования, на котором вы пишете. Таблицы из базы данных могут быть представлены обычных классов. В Django они называются моделями.

ORM. Модели

Table `Car`

brand	name	color
Text	Text	Text

Table `Person`

name	birthday	gender	car
Text	Text	Text	Integer

```
from django.db import models

class Car(models.Model):
    brand = models.TextField()
    name = models.TextField()
    color = models.TextField()

class Person(models.Model):
    name = models.TextField()
    birthday = models.DateField()
    gender = models.TextField()
    car = models.ForeignKey('Car', on_delete=models.CASCADE)
```

Связи между моделями

```
class Article(models.Model):
    ...
    reporter = models.ForeignKey(Reporter, on_delete=models.CASCADE)
```

- `models.ForeignKey` — поле связи многие к одному. Принимает позиционный аргумент: класс связанной модели.
- `models.CASCADE` — автоматически удаляет строку из зависимой таблицы, если удаляется связанная строка из главной таблицы.

ORM. Поля

Можно заметить, что название классов для полей заканчиваются на `Field`

- BooleanField;
- CharField;
- DateField / DateTimeField;
- IntegerField;
- FloatField.

ORM. Модели

ORM — это универсальный инструмент по работе с данными.

```
# получаем всех людей
persons = Person.objects.all()

# получаем название машины
# первого человека из выборки
persons[0].car.name
```

На самом деле тут будет два запроса, и Django знает, как их правильно сформировать.

ORM. Оптимальные запросы к базе

Получаем необходимые данные одним запросом

```
Person.objects.all().select_related('car')
```

`select_related` позволяет соединять две модели. Работает только для связи «один к одному» и «многие к одному».

Запросы к БД с помощью ORM

Получение объекта по `id`

```
person = Person.objects.get(id=1)
```

Фильтрация по полям в запросах

Для выбора нужного объекта из БД:

```
Person.objects.filter(
     birthday__gt=datetime.date().today() –
datetime.timedelta(days=365*18)
).exclude(
     is_deleted=True
)
```

- startswith — строка начинается с указанной подстроки
- gt / gte — больше (greater than) / больше либо равно (greater than or equal)
- lt и lte — меньше (less than) / меньше либо равно (less than or equeal)
- in — входит в список, например status__in=(0, 1, 2)
- isnull ...

и много других, полный список тут.

Создание и сохранение объекта в БД

```
# Создание объекта в Python и затем сохранение в БД
p = Person(first_name='John',last_name='Doe')
p.save()

# Сохранение объекта в БД
p = Person.objects.create(first_name="John", last_name="Doe")
```

Для сохранения изменений нужно вызвать метод `.save()`

Создание БД проекта

Для начала нужно создать базу данных.
По умолчанию при создании проекта прописана SQLite. `settings.py`

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'your_project_specific_db_name',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

В реальных веб-проектах её не используют. SQLite используется для встраиваемых систем и прикладного ПО. (например Firefox). Поэтому мы используем postgresql.

После написания моделей необходимо внести изменения в БД. Для этого используются миграции.

Создание Миграции

Модели в Django позволяет автоматически создавать нужные таблицы или вносить изменения в БД.

```
$ python manage.py makemigrations

Migrations for 'car_enthusiasts':
  car_enthusiasts/migrations/0001_initial.py
    - Create model Car
    - Create model Person
# основная команда для применения всех непримененных миграций
$ python manage.py migrate
```

```
# откат изменения на нужную точку
python manage.py migrate car_enthusiasts 0001
# откат всех изменений
python manage.py migrate car_enthusiasts zero
```

SQL для создании таблицы

Создание схемы данных тоже происходит с помощью запроса к базе данных. Для изучения вы можете посмотреть какой запрос сформирует Django:

```
# Указываем приложение и номер миграции
# которая была создана командой makemigrations
$ python manage.py sqlmigrate car_enthusiasts 0001
```

SQL для создании таблицы

```
BEGIN;
--
-- Create model Car
--
CREATE TABLE "car_enthusiasts_car" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "brand" text NOT NULL,
    "name" text NOT NULL,
    "color" text NOT NULL
);
--
-- Create model Person
--
CREATE TABLE "car_enthusiasts_person" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "name" text NOT NULL,
    "birthday" date NOT NULL,
    "gender" text NOT NULL,
    "car_id" integer NOT NULL REFERENCES "car_enthusiasts_car" ("id")
DEFERRABLE INITIALLY DEFERRED
);
CREATE INDEX "car_enthusiasts_person_car_id_40e2f062"
  ON "car_enthusiasts_person" ("car_id");
COMMIT;
```

Внутри миграции

```
from django.db import migrations, models

class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Phone',
            fields=[
                ('id', models.AutoField(primary_key=True, serialize=False)),
                ('name', models.CharField(max_length=64, verbose_name='Модель телефона')),
                ('price', models.IntegerField(verbose_name='Цена')),
                ('image', models.ImageField(upload_to='', verbose_name='Изображение')),
                ('slug', models.SlugField(blank=True, unique=True, verbose_name='URL')),
            ],
            options={
                'verbose_name': 'телефон',
                'verbose_name_plural': 'Телефон',
            },
        ),
    ]
```

Модели и View

В Django есть Class Based View. Это view написанные в виде классов.

CBV реализуют базовые потребности, например: отображение списка объектов (ListView), отображение информации об объекте (DetailView).

У них есть поле `model`. Можно его задать и django сделает за вас все остальные подготовительные части, которые вы в любом месте сможете переопределить в случае необходимости.

```
class CarList(ListView):
    model = car
```

Другие поля можно посмотреть в исходниках или здесь: http://ccbv.co.uk/

Управление БД

Для просмотра содержимого БД можно установить стороннее приложение. Например, `PGAdmin`.

Управление БД из админки Django

Админка Django инструмент для модераторов и админов. Делает редактирование моделей удобным и позволяет простым пользователям менять сущности в базе данных без необходимости писать Python-код и sql-запросы.

В несколько строк можно получить средство для редактирования БД. Для подключения модели БД в админке нужно создать view. Пишется в файле `admin.py`.

Управление БД из админки Django

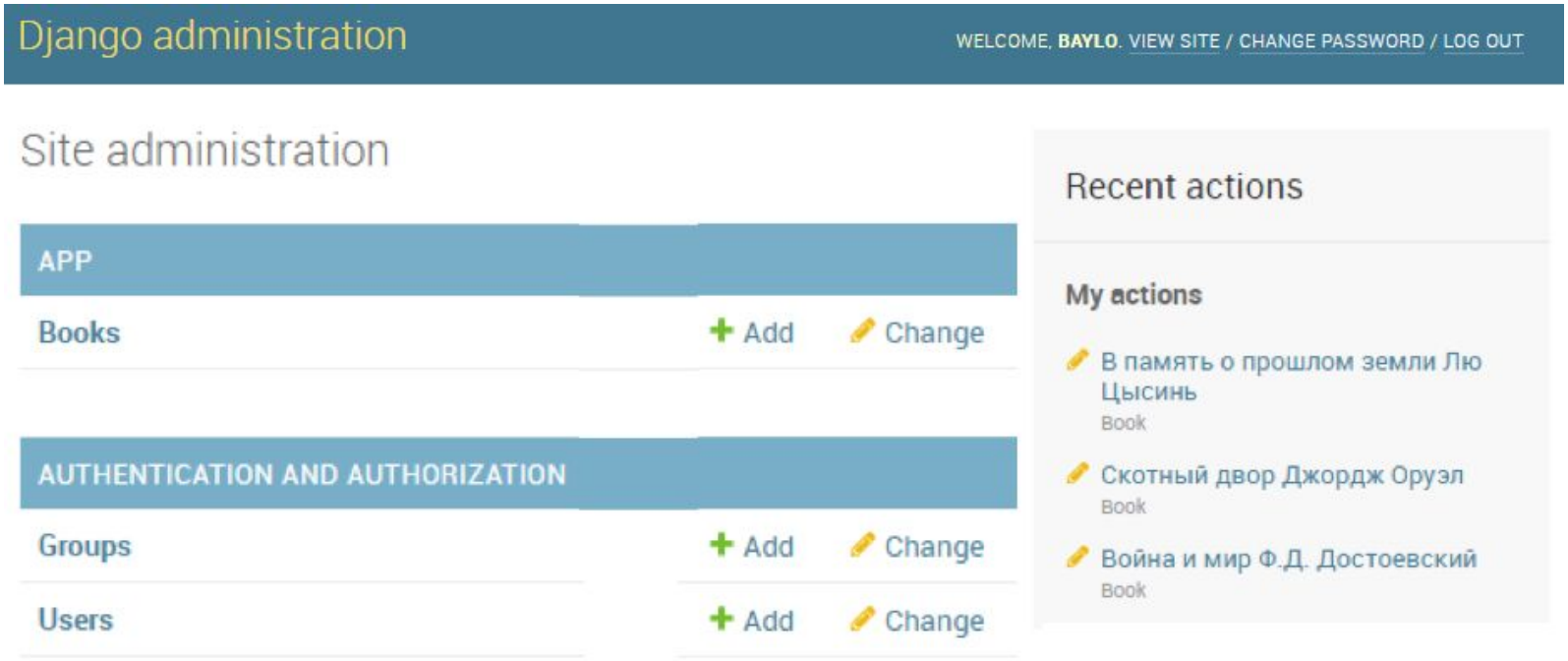
```
from django.contrib import admin
from .models import Person, Car

# Регистрация страницы в админке через декоратор
@admin.register(Person)
class PersonAdmin(admin.ModelAdmin):
    pass


# Регистрация страницы в адменке через вызов функции
class CarAdmin(admin.ModelAdmin):
    pass

admin.site.register(Car, CarAdmin)
```

Базовые отображения в админке

На стартовой странице административной части отображаются следующие элементы:

- app — название приложения, которое мы создаем командой startapp app
- Books — название зарегистрированной модели в файле admin.py
- AUTHENTICATION AND AUTHORIZATION — название приложения для управления пользователями
- Users — здесь можно управлять зарегистрированными пользователями
- Groups — здесь можно управлять группами пользователей
- Add — добавить новый элемент
- Recent actions — последние действия в админке

Базовые отображения в админке



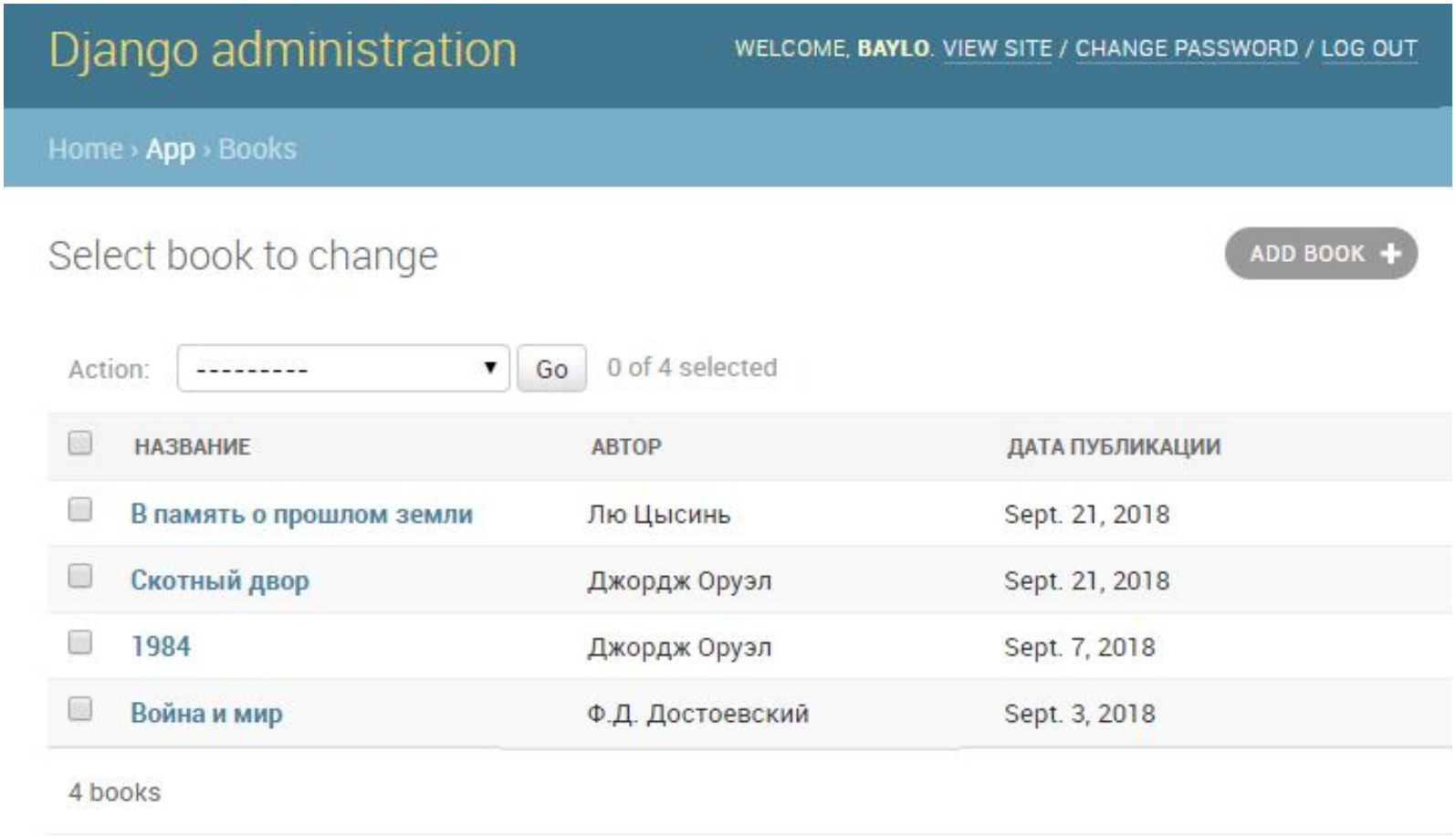
Базовые отображения в админке

При нажатии на отображение модели Books или нажатием на Change напротив соответствующей модели открывается детальная страница данной модели, на которой отображаются все элементы этой модели. Какие именно поля будут отображаться — можно настроить в файле `admin.py`, например:

```
class BookAdmin(admin.ModelAdmin):
    list_display = ('name', 'author', 'pub_date', )
```

Здесь же можно добавлять новые элементы и удалять старые. Также, при нажатии на любой отдельный элемент, откроется страница с детализацией, на которой можно изменять этот элемент.

Базовые отображения в админке



Итоги

В ходе лекции мы посмотрели как:

- создавать и редактировать структуру базы данных,
- генерировать запросы к данным,
- получить редактор данных с системой доступа и прав,
- быстро создавать страницы со списками объектах и детальной информацией об объекте.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Максим Филипенко

 mfiliprnko.yandex.ru |  prokaktus |  prokaktus