Работа с ORM

Александр Мужев

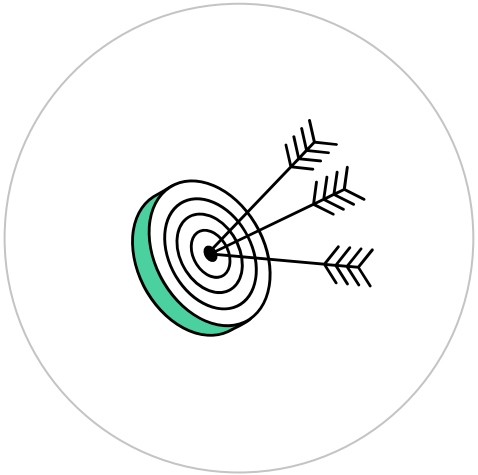
Александр Мужев

О спикере:

- Старший инженер по тестированию в компании "ООО Альянс АйтиТехнолоджи"
- Эксперт продвинутого курса "Инженер по тестированию: с нуля до middle" в образовательной платформе ООО Нетология
- Преподаватель QA (очная форма) по направлению ручное и автоматизированное тестирование ПО в компьютерной академии ТОР
- Репетитор по ручному и автоматизированному тестированию ПО (расширенный курс). Провожу индивидуальные занятия со студентами.
- QA Full Stack-фрилансер
- Неоднократно организовал весь процесс тестирования с нуля
- Опыт работы в тестировании банковских продуктов и приложений сферы медицинских услуг.

Цели занятия

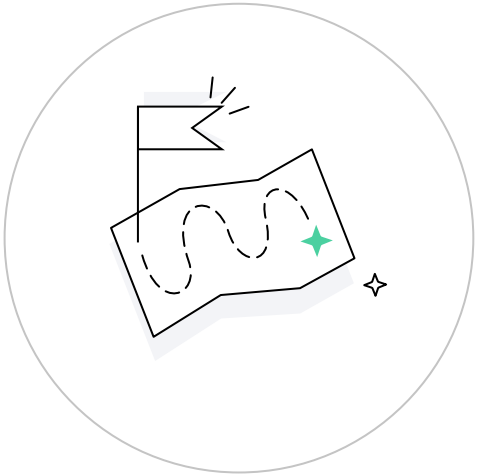
- Узнать, как подключить СУБД к Django-проекту
- Разобраться, что такое ORM и за что отвечает Django ORM
- Научиться создавать модели и выполнять миграции в Django
- Посмотреть, как управлять записями в таблицах через административную панель Django
- Познакомиться с типами связей между таблицами в Django
- Научиться выполнять запросы к БД при помощи ORM



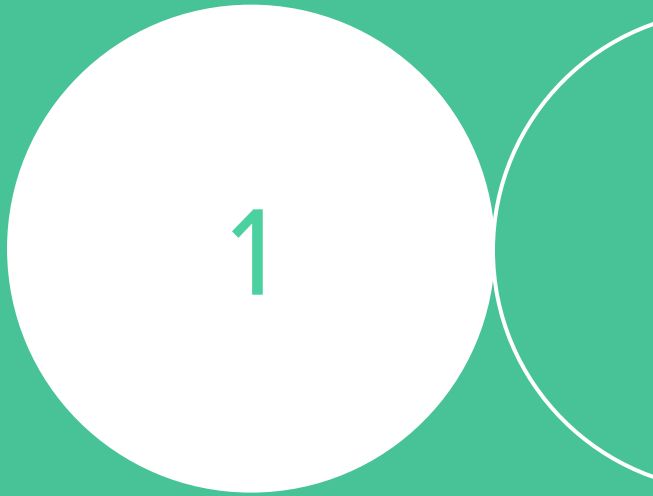
План занятия

1. Подключение БД к Джанго-проекту
2. ORM. Как связать SQL и Django
3. Модели данных. Поля, типы и свойства полей
4. Административная панель
5. Построение запросов
6. Выборка данных. Queryset, lookup
7. Связи между таблицами
8. Итоги
9. Домашнее задание

*Нажмите на нужный раздел для перехода



Подключение БД к Django-проекту



Вспоминаем прошлые модули

Вопрос: Что такое БД?

Вспоминаем прошлые модули

Вопрос: Что такое БД?

Ответ: База данных — это упорядоченный набор структурированной информации или данных, которые обычно хранятся в электронном виде в компьютерной системе.

Данные вместе с СУБД, а также приложения, которые с ними связаны, называются системой баз данных (или базой данных)

Вспоминаем прошлые модули

Вопрос: Из чего состоят реляционные БД?

Вспоминаем прошлые модули

Вопрос: Из чего состоят реляционные БД?

Ответ: Реляционные БД состоят из таблиц.

Таблица — основной объект базы данных, который предназначен для хранения данных в структурированном виде.

Каждая таблица БД состоит из строк и столбцов для хранения информации об однотипных объектах системы

Подключение Django к БД

Для подключения Django к БД по умолчанию использует SQLite.

SQLite – легковесная и не требует сложной настройки

SQLite имеет ограниченный функционал и не подходит для использования в production

Подключение Django к БД

В Django из коробки реализована поддержка следующих СУБД:

- PostgreSQL
- MySQL
- Maria DB
- Oracle Database.

Также имеется возможность подключать другие SQL и NoSQL СУБД при помощи сторонних библиотек

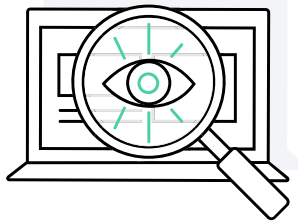
Подключение Django к PostgreSQL

1. Создать БД
2. Установить в виртуальное окружение проекта драйвер СУБД для Python (модуль **psycopg2-binary**)
3. Изменить настройки БД в **settings.py**

```
DATABASES = {
    “default”: {
        “ENGINE”: “django.db.backends.postgresql”,
        “NAME”: “<название БД>”
        “USER”: “<пользователь БД>”,
        “PASSWORD”: “<пароль пользователя>”,
        “HOST”: “<адрес хоста>”, #по умолчанию localhost
        “PORT”: “<номер порта>” #по умолчанию 5432
    }
}
```

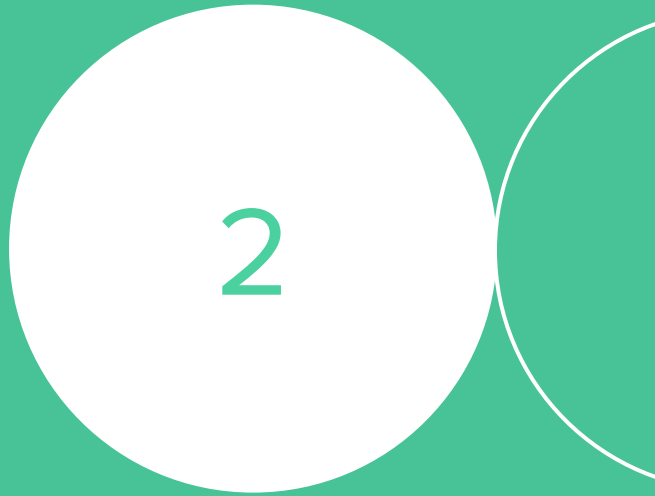
Демонстрация работы

Создание Django-проекта, подключение базы данных



ORM

Как связать SQL и Django

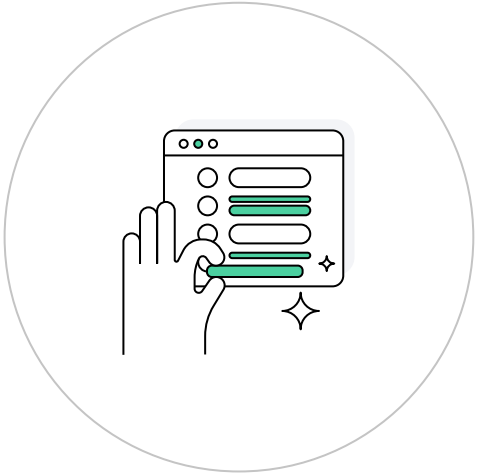


ORM

Мы подключили БД.

Как работать с БД? Каким образом выполнять запросы?

Для этого будем использовать технологию **ORM**



ORM (Object-relational mapping)

Прослойка между SQL и языком программирования.

Позволяет:

- работать с таблицами с помощью классов Python
- избежать написания SQL-кода
- взаимодействовать с БД при помощи вашего языка программирования

Django ORM

Django предоставляет свой Django ORM, который использует подход Active Record.

Active Record – схема, при которой каждой таблице в БД соответствует только один класс (модель).

Модель отвечает за реализацию бизнес-логики и за выполнение запросов в БД

Что делает Django ORM?

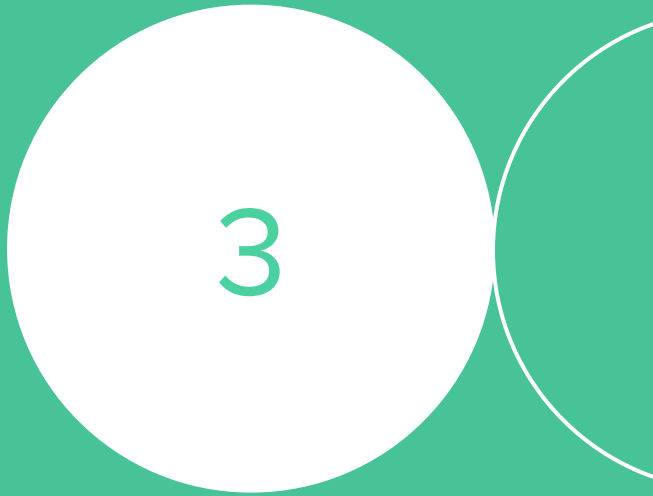
Django ORM отвечает за изменение схемы БД:

- добавление, изменение, удаление таблиц
- организацию связей между таблицами и др.

Для внесения изменений используется механизм **миграций**

Модели данных

Поля, типы и свойства полей



Модель

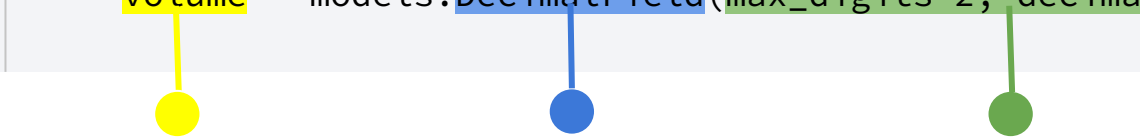
Python-класс, выражающий таблицу в БД. Как правило, каждая модель отображается в одну таблицу БД

Свойства моделей

- Каждая модель представляет собой класс Python, который является подклассом **django.db.models.Model**
- Каждый атрибут модели представляет поле базы данных
- При этом Django предоставляет автоматически сгенерированный API доступа к базе данных

Поля модели

```python
from django.db import models


class Car(models.Model):
    brand = models.CharField(max_length=100)
    model = models.CharField(max_length=100)
    …
    year = models.IntegerField()
    color = models.CharField(max_length=20)
    volume = models.DecimalField(max_digits=2, decimal_places=1)
```



Наименование поля

Тип поля

Свойства поля

Поля модели в таблице

ORM отражает вышеуказанную модель в таблицу БД вида:

<app>_car					
brand	**model**	**...**	**year**	**color**	**volume**
VARCHAR(100)	Нет		INTEGER	VARCHAR(20)	DECIMAL(2, 1)

Поля модели в таблице

ORM отражает вышеуказанную модель в таблицу БД вида:

<app>_car					
brand	**model**	**...**	**year**	**color**	**volume**
VARCHAR(100)	Нет		INTEGER	VARCHAR(20)	DECIMAL(2, 1)

Название таблицы по умолчанию формируется в формате:
<название_приложения>_<имя_модели>. Таблице можно задать **желаемое имя**.

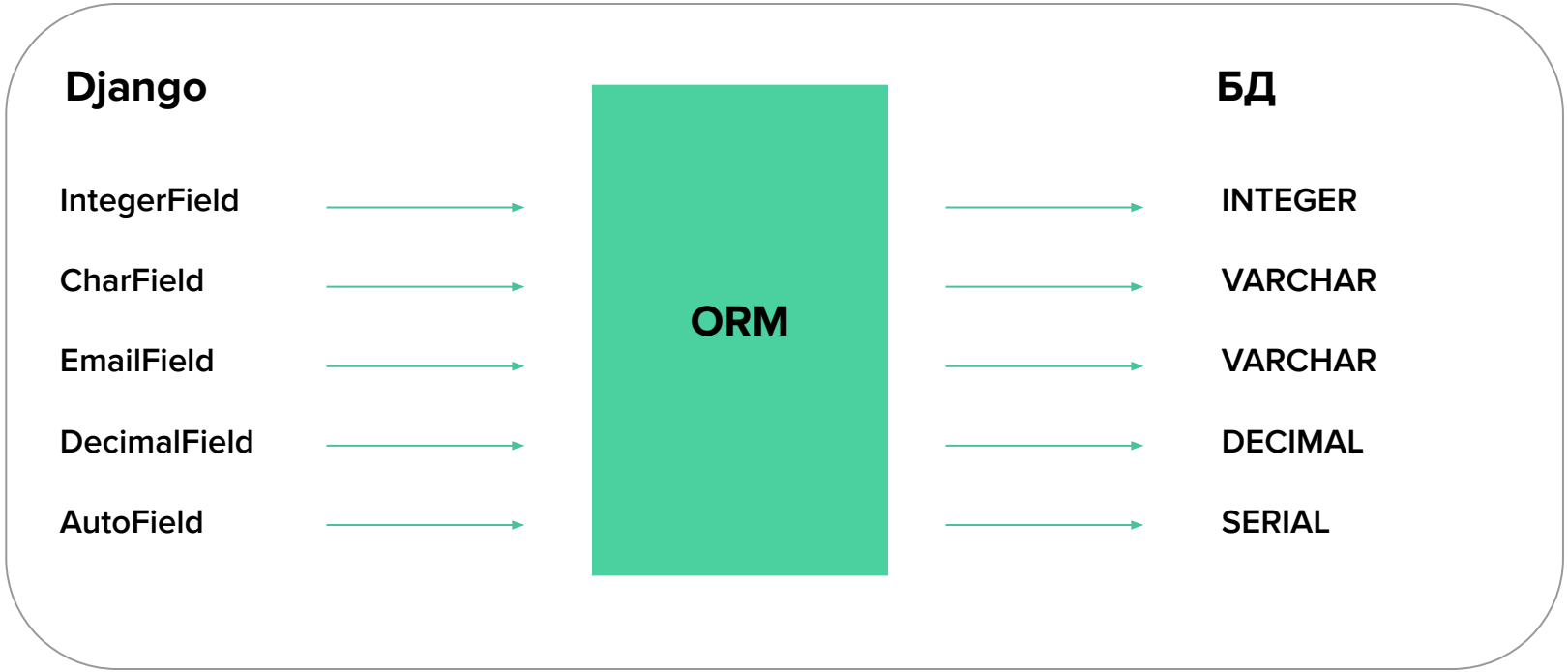
Столбцы в таблице получают те же имена, что и поля в модели (за исключением полей для отношений). Каждому типу поля в модели соответствует определённый тип в БД. Переносятся все свойства, указанные в поле модели

Типы полей

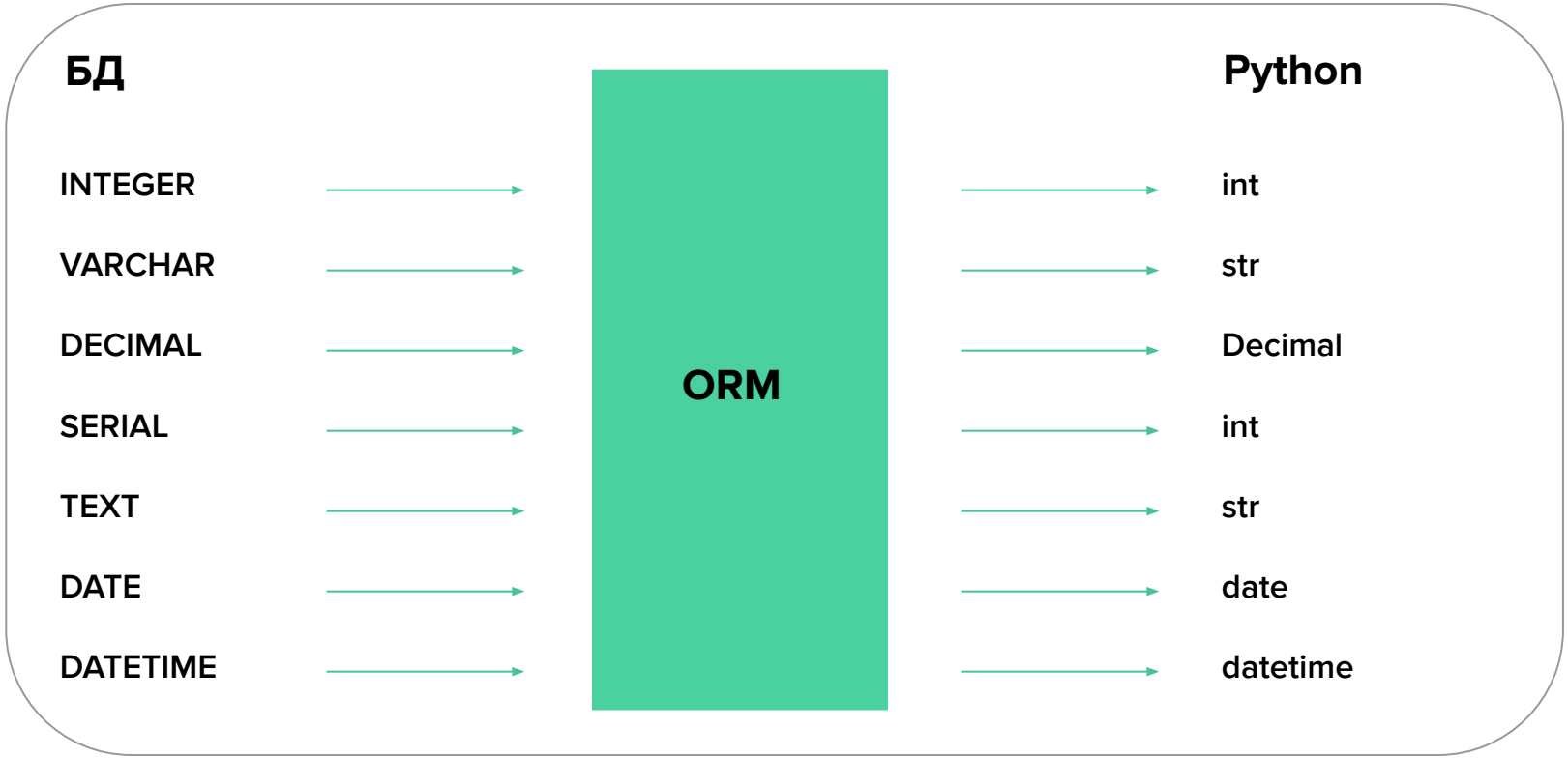
Модели предоставляют большое количество типов полей: числовые, строчные, время, дата и др. Эти классы **содержат в названии слово Field**: IntegerField, CharField, TextField, EmailField, FileField и т.д.

Помимо того, что каждый тип поля имеет свой тип в БД, он также реализует проверки на входящие значения и преобразование в соответствующий тип Python

Типы полей в Django и БД



Преобразование данных в Python объекты



Миграции

Мы создали модель, но как сделать так, чтобы создалась соответствующая таблица в БД и можно было сохранять данные?

Для этого мы используем **миграции**

Миграции

Механизм для автоматического управления изменениями в схеме базы данных.

Это важный инструмент разработки веб-приложений, поскольку позволяет легко и контролируемо изменять структуру базы данных в соответствии с изменениями в моделях Django

Преимущества использования миграций

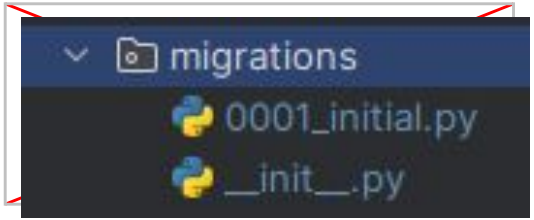
1. Автоматизация изменений
2. Версионирование базы данных
3. Переносимость приложения
4. Облегчение совместной работы
5. Безопасность данных

Создание миграций

Чтобы создать миграцию, введите команду:

```
python manage.py makemigrations
```

После этого, в пакете migrations приложения появится файл миграции:



В файле миграции записываются изменения, которые произошли в моделях: **создание/удаление модели, добавление/удаление/изменение полей, изменение типа поля, добавление/удаление ограничений** и т.д.

Пример файла миграции

```python
import django.core.validators
from django.db import migrations, models
import django.db.models.deletion


addirossi
class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Product',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('title', models.CharField(max_length=60, unique=True)),
                ('description', models.TextField(blank=True, null=True)),
            ],
        ),
        migrations.CreateModel(
            name='Stock',
            fields=[
                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
```

Применение миграций

Чтобы применить в БД изменения, сохранённые в файле миграций, необходимо выполнить команду:

```
python manage.py migrate
```

В таком случае автоматически применятся все миграции во всех приложениях.

Чтобы применить изменения для конкретной модели, можно воспользоваться командой:

```
python manage.py migrate <название_приложения>
```

Откат миграций

Откат к любой предыдущей миграции для конкретного приложения:

```
python manage.py migrate <название_приложения> <название_миграции>
```

Откат всех миграций для конкретного приложения:

```
python manage.py migrate <название_приложения> zero
```

Откат всех миграций для всех приложений:

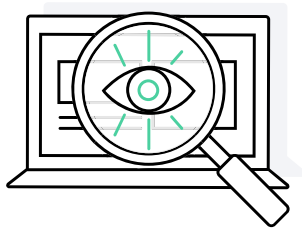
```
python manage.py migrate zero
```

Просмотр истории миграций:

```
python manage.py showmigrations
```

Демонстрация работы

Создание моделей, выполнение миграций



Административная панель

Заполнение БД

4

Административная панель Django

Встроенный интерфейс, который облегчает управление данными приложения.

Панель создаётся автоматически на основе моделей данных и предоставляет удобный способ взаимодействия с базой данных

Алгоритм использования административной панели Django

Шаг 1. Создание суперпользователя

Прежде чем начать использовать административную панель, необходимо создать суперпользователя.

Выполним следующую команду в терминале:

```
python manage.py createsuperuser
```

Алгоритм использования административной панели Django

Шаг 2. Регистрация моделей

Откроем файл **admin.py** в Django-приложении и зарегистрируем модели, которыми будем управлять в административной панели

```
# admin.py

from django.contrib import admin
from .models import YourModel

admin.site.register(YourModel)
```

Алгоритм использования административной панели Django

Шаг 3. Запуск сервера разработки

Убедимся, что сервер разработки запущен:

```
python manage.py runserver
```

Алгоритм использования административной панели Django

Шаг 4. Вход в административную панель

Откроем браузер и перейдём по адресу **http://127.0.0.1:8000/admin/**.

Войдём, используя учётные данные суперпользователя

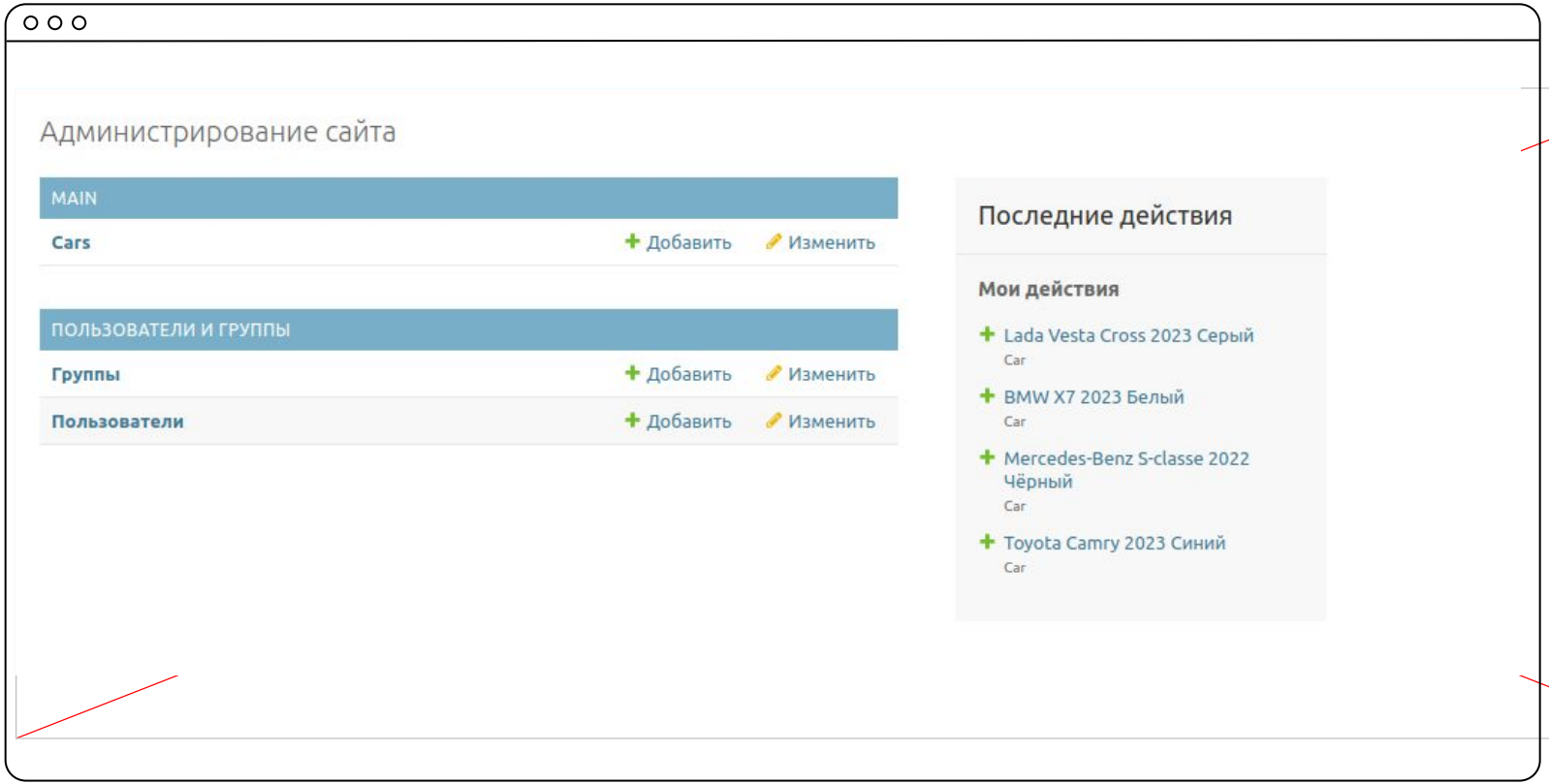
Алгоритм использования административной панели Django

Шаг 5. Использование административной панели

Теперь можно управлять данными приложения через административную панель.

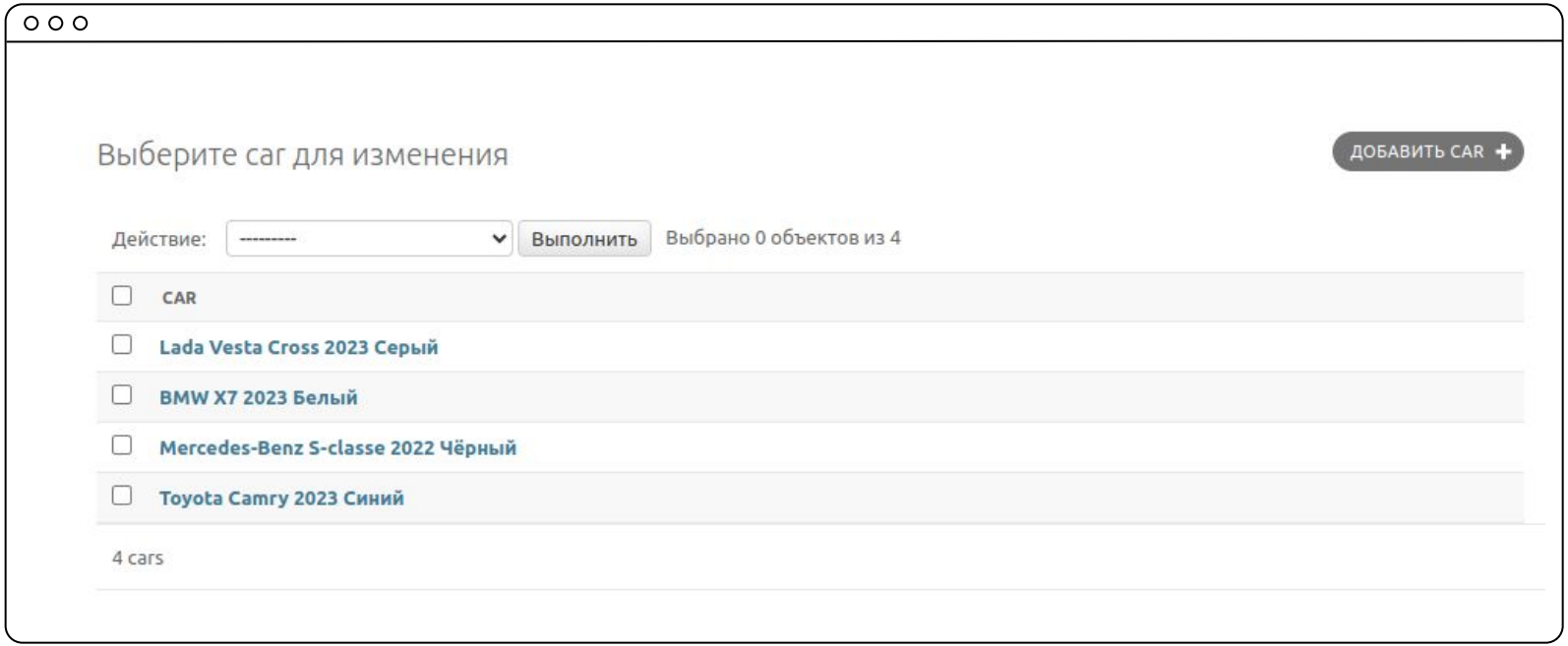
Вы увидите зарегистрированные модели. Для каждой модели будет доступна форма для добавления, изменения и удаления записей

Пример файла миграции



Отображение списка записей

При переходе в отображение модели можно увидеть все записи, которые имеются в БД



Создание, редактирование и удаление записей

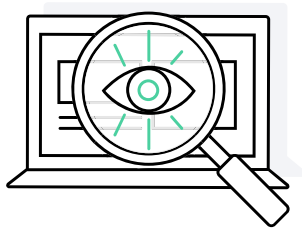
Чтобы **добавить** новую запись в БД, нажмите на кнопку «Добавить».

Чтобы перейти на страницу **редактирования** записи, нажмите на любую запись.

На этой же странице расположена кнопка «Удалить», с помощью которой можно **удалить** текущую запись

Демонстрация работы

Настройка и использование административной панели



Построение запросов

Менеджер модели

Менеджер модели

Для выполнения запросов в БД (создание, выборка, обновление, удаление записей и т.д.) модель использует специальный объект – **менеджер**.

Django по умолчанию предоставляет свойство **objects** для каждой модели. Если вы не определите свой менеджер, Django будет использовать objects по умолчанию

Отображение списка записей

```python
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    # Собственный менеджер модели
    objects = models.Manager()

    # Дополнительные методы менеджера модели
    def published_books(self):
        return self.objects.filter(status='published')
```

В примере, objects – это менеджер модели по умолчанию. Вы можете добавлять собственные методы к менеджеру для выполнения запросов.

В данном случае, добавлен метод published_books, который возвращает все опубликованные книги

Создание одного объекта

Создание записей в Django может быть выполнено несколькими способами в зависимости от того, как настроены модели и менеджеры. Вот несколько примеров:

```
from myapp.models import MyModel

# Создание записи при помощи метода create()
MyModel.objects.create(field1="value1", field2="value2")

# Создание записи путём создания экземпляра модели
my_obj = MyModel(field1="value1", field2="value2")
my_obj.save() # сохранение записи в БД
```

Создание списка объектов

Помимо создания одного объекта, Вы также можете создавать сразу несколько объектов методом **bulk_create()**

```
from myapp.models import MyModel


# Список экземпляров модели для создания
records_to_create = [
    MyModel(field1="value1", field2="value2"),
    MyModel(field1="value3", field2="value4"),
    # Добавьте другие экземпляры по мере необходимости
]

# Массовое создание записей
MyModel.objects.bulk_create(records_to_create)
```

Получение объектов

Получение объектов из базы данных также осуществляется с использованием менеджеров моделей

```
from myapp.models import MyModel


# Получение всех объектов
all_objects = MyModel.objects.all()

# Получение первого объекта, где поле field1 равно 'value1'
single_object = MyModel.objects.get(field1='value1')

# Получение объектов, где поле field1 равно 'value1'
filtered_objects = MyModel.objects.filter(field1='value1')
```

Обновление объектов

```
from myapp.models import MyModel


# Массовое обновление всех объектов, где поле field1 равно 'old_value'
MyModel.objects.filter(field1='old_value').update(field1='new_value')

# Получение объекта для обновления
obj_to_update = MyModel.objects.get(pk=1)

# Обновление поля field1
obj_to_update.field1 = 'new_value'
obj_to_update.save()
```

Удаление объектов

```
from myapp.models import MyModel


# Удаление всех объектов из таблицы
MyModel.objects.all().delete()

# Удаление всех объектов, где поле field1 равно 'value'
MyModel.objects.filter(field1='value').delete()

# Получение объекта для удаления
obj_to_delete = MyModel.objects.get(pk=1)
# Удаление объекта
obj_to_delete.delete()
```

Выборка данных

QuerySet, фильтры(lookup)

Получение одного объекта

Чтобы получить один объект из базы данных в Django, можно применить метод **get()** менеджера модели, например

```
from myapp.models import MyModel

try:
    # Получение объекта, где поле field1 равно 'value1'
    single_object = MyModel.objects.get(field1='value1')
except MyModel.DoesNotExist:
    # Обработка ситуации, когда объект не найден
    print("Объект не найден")
except MyModel.MultipleObjectsReturned:
    # Обработка ситуации, когда найдено более одного объекта
    print("Найдено более одного объекта")
else:
    # Обработка найденного объекта
    print("Найден объект:", single_object)
```

Получение одного объекта

Если необходимо получить первый объект, который соответствует условию фильтрации, можно использовать метод **filter()** и затем взять первый элемент из результата

```
from myapp.models import MyModel


# Получение первого объекта, где поле field1 равно 'value1'
single_object = MyModel.objects.filter(field1='value1').first()

if single_object:
    print("Найден объект:", single_object)
else:
    print("Объект не найден")
```

В этом случае, если объект не найден, **single_object** будет равен **None**

Получение списка объектов

Чтобы получить список объектов из базы данных в Django, можно использовать методы **all()**, **filter()** и другие методы менеджера модели

```
from myapp.models import MyModel

# Получение всех объектов
all_objects = MyModel.objects.all()

# Получение объектов, где поле field1 равно 'value1'
filtered_objects = MyModel.objects.filter(field1=”value1”)

# Получение объектов, где поле field1 равно 'value1' ИЛИ поле field2 равно
'value2'
complex_objects = MyModel.objects.filter(field1=”value1”) |
MyModel.objects.filter(field2=”value2”)

#фильтрация по нескольким условиям одновременно
filtered_objects = MyModel.objects.filter(field1=”value1”, field3=”value3”)
```

Фильтрация данных

Для фильтрации объектов из базы данных, можно использовать метод **filter()**. Этот метод принимает аргументы для определения условий фильтрации

```
from myapp.models import MyModel

# Получение объектов, где поле field1 равно 'value1'
filtered_records = MyModel.objects.filter(field1=”value1”)

# Получение объектов, где поле field1 равно 'value1' и поле field2 равно
'value2'
filtered_records_multiple = MyModel.objects.filter(field1=”value1”,
field2=”value2”)

# Получение объектов, где поле field1 содержит подстроку 'text'
contains_records = MyModel.objects.filter(field1__contains=”text”)

# Исключение объектов, где поле field1 равно 'value1'
excluded_records = MyModel.objects.exclude(field1=”value1”)
```

Фильтрация данных

Вот ещё некоторые варианты фильтрации

```
from django.db.models import F, Q
from myapp.models import MyModel


# Получение объектов, где поле field1 равно 'value1' ИЛИ поле field2 равно
'value2'
complex_records = MyModel.objects.filter(Q(field1='value1') |
Q(field2='value2'))

# Получение объектов, где поле created_at больше определенной даты
filtered_by_date = MyModel.objects.filter(created_at__gt='2022-01-01')

# Получение объектов, где поле field1 равно полю field2
records_equal_fields = MyModel.objects.filter(field1=F('field2'))
```

Фильтры по полям (lookup)

lookup (или поиск) – это механизм, который позволяет создавать более сложные условия фильтрации. Lookup'ы добавляются к именам полей в методах фильтрации для определения точных условий поиска

```
from myapp.models import MyModel

# Получение объекта, где поле field1 точно соответствует 'value1'
result = MyModel.objects.get(field1__exact='value1')

# Получение объекта, где поле field1 точно соответствует 'value1' без учёта регистра
result = MyModel.objects.get(field1__iexact='value1')

# Получение объектов, где поле field1 содержит подстроку 'text'
result = MyModel.objects.filter(field1__contains='text')

# Получение объектов, где поле field1 начинается с 'prefix'
result_startswith = MyModel.objects.filter(field1__startswith='prefix')

# Получение объектов, где поле field1 заканчивается на 'suffix'
result_endswith = MyModel.objects.filter(field1__endswith='suffix')
```

Фильтры по полям (lookup)

И ещё некоторые варианты lookup

```
from myapp.models import MyModel

# Получение объектов, где поле field1 включено в список ['value1', 'value2']
result = MyModel.objects.filter(field1__in=['value1', 'value2'])

# Получение объектов, где поле field1 находится в диапазоне от 10 до 20
result = MyModel.objects.filter(field1__range=[10, 20])

# Получение объектов, где поле field1 равно null
result = MyModel.objects.filter(field1__isnull=True)

# Получение объектов, где поле field1 больше 10
result_gt = MyModel.objects.filter(field1__gt=10)

# Получение объектов, где поле field1 больше или равно 10
result_gte = MyModel.objects.filter(field1__gte=10)

# Получение объектов, где поле field1 меньше 10
result_lt = MyModel.objects.filter(field1__lt=10)

# Получение объектов, где поле field1 меньше или равно 10
result_lte = MyModel.objects.filter(field1__lte=10)
```

Сортировка

Для сортировки данных, можно использовать метод **order_by()**. Метод **order_by()** принимает одно или несколько полей, по которым будет производиться сортировка.

По умолчанию сортировка происходит по возрастанию, но можно указать порядок с помощью префикса **"-"** для поля (убывающий порядок)

```
from myapp.models import MyModel


# Сортировка по полю field1 в возрастающем порядке
sorted_records = MyModel.objects.all().order_by('field1')

# Сортировка по полю field1 в убывающем порядке
sorted_records_desc = MyModel.objects.all().order_by('-field1')

# Сортировка по полю field1 в возрастающем порядке и по полю field2 в убывающем порядке
sorted_records_multiple = MyModel.objects.all().order_by('field1', '-field2')

# Сортировка по полю field1, игнорируя регистр
sorted_records_case_insensitive = MyModel.objects.all().order_by('field1__iexact')
```

Подсчёт количества записей

Для подсчёта количества записей, удовлетворяющих определенным условиям, можете использовать метод **count()** на объекте QuerySet. Вот несколько примеров:

```
from myapp.models import MyModel


# Подсчёт всех записей в модели
total_records = MyModel.objects.all().count()

# Подсчёт записей, где поле field1 равно 'value1'
filtered_records = MyModel.objects.filter(field1='value1').count()

# Подсчёт записей, где поле field1 равно 'value1' ИЛИ поле field2 равно 'value2'
complex_records = MyModel.objects.filter(field1='value1') |
MyModel.objects.filter(field2='value2').count()

# Подсчёт записей, где поле field1 содержит подстроку 'text'
contains_records = MyModel.objects.filter(field1__contains='text').count()
```

Агрегация

Для агрегации данных, например, подсчёта, суммирования или вычисления среднего значения можно использовать **функции агрегации QuerySet и аннотаций**

```
from myapp.models import MyModel


# Суммирование значений поля 'amount'
total_amount = MyModel.objects.all().aggregate(Sum('amount'))['amount__sum']

# Вычисление среднего значения поля 'price'
average_price = MyModel.objects.all().aggregate(Avg('price'))['price__avg']

# Поиск максимального значения поля 'score'
max_score = MyModel.objects.all().aggregate(Max('score'))['score__max']

# Поиск минимального значения поля 'quantity'
min_quantity = MyModel.objects.all().aggregate(Min('quantity'))['quantity__min']
```

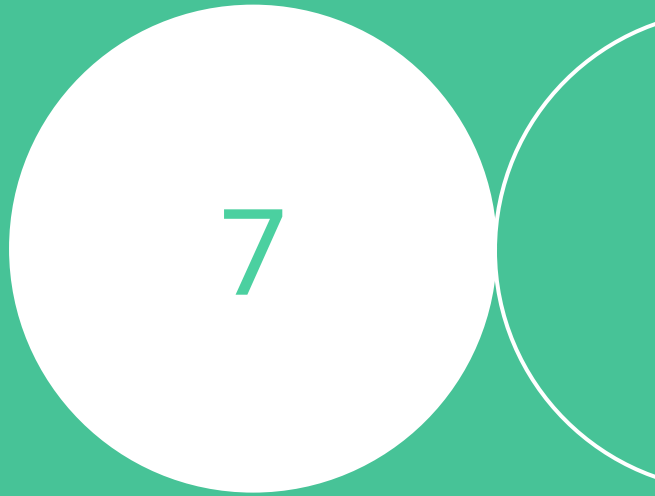
Агрегация

Ещё несколько примеров агрегации:

```
from myapp.models import MyModel


# Группировка по полю 'category' и подсчёт количества записей в каждой группе
category_counts =
MyModel.objects.values('category').annotate(record_count=Count('id'))

# Увеличение значения поля 'quantity' на значение поля 'bonus_quantity'
result = MyModel.objects.annotate(total_quantity=F('quantity') +
F('bonus_quantity')).values('total_quantity')

# Подсчёт количества записей, удовлетворяющих одному из двух условий
complex_aggregation = MyModel.objects.filter(Q(price__gte=100) |
Q(quantity__gte=10)).count()
```

Связи между таблицами

Типы связей, запросы связанных объектов



Типы связей

В Django связи между таблицами в базе данных определяются с использованием различных типов полей модели.

Основные типы связей:

- один к одному
- многие к одному
- многие ко многим
- самореференцированная связь

Один к одному (One to one)

Связь «один к одному» задаётся типом поля **OneToOneField**

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=100)

class Passport(models.Model):
    person = models.OneToOneField(Person,
on_delete=models.CASCADE)
    passport_number = models.CharField(max_length=20)
```

Многие к одному (Many to one)

Связь «многие к одному» задаётся типом поля **ForeignKey**

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author,
on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
```

Многие ко многим (Many to many)

Связь «многие ко многим» задаётся типом поля **ManyToManyField**

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100)

class Course(models.Model):
    name = models.CharField(max_length=100)
    students = models.ManyToManyField(Student)
```

Самореференцированная связь (Self-Referential)

Этот тип связи является частным случаем связи, когда один объект модели ссылается на другой объект этой же модели

```
from django.db import models

class Employee(models.Model):
    name = models.CharField(max_length=100)
    manager = models.ForeignKey('self',
on_delete=models.SET_NULL, null=True, blank=True)
```

Итоги занятия

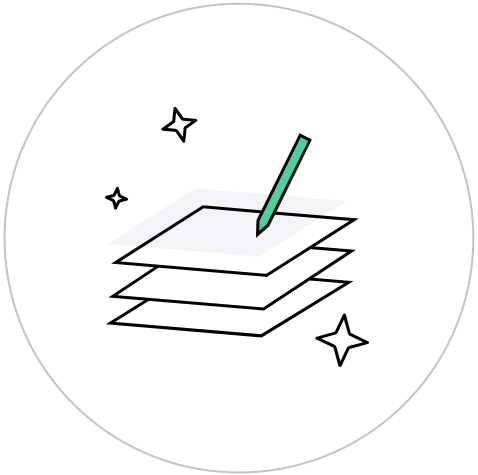
Сегодня мы

- Научились подключать СУБД к Django-проекту
- Узнали, что такое ORM, модели и попробовали применить изменения в БД
- Поняли, как создавать записи в БД при помощи административной панели
- Рассмотрели связи между таблицами в Django
- Научились выполнять запросы в БД с помощью ORM

Домашнее задание

Давайте посмотрим ваше домашнее задание.

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Работа с ORM

Александр Мужев