1. Создание моделей

Модели в Django описывают структуру используемых данных. Используемые в программе данные хранятся в базах данных, и с помощью моделей как раз осуществляется взаимодействие с базой данных.

Для работы с базами данных в проекте Django в файле **settings.py** определен параметр **DATABASES**, который по умолчанию выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | DATABASES = {      'default': {          'ENGINE': 'django.db.backends.sqlite3',          'NAME': os.path.join(BASE\_DIR, 'db.sqlite3'),      }  } |

Конфигурация используемой базы данных в данном случае складывается из двух параметров. Параметр **ENGINE** указывает на используемый движок для доступа к БД. В данном случае это встроенный пакет django.db.backends.sqlite3. Второй параметр - **NAME** указывает на путь к базе данных. После первого запуска проекта в нем по умолчанию будет создан файлdb.sqlite3, который собственно и будет использоваться в качестве базы данных.

**Установка СУБД МySQL: pip install mysql-python**

При создании приложения по умолчанию в его каталог добавляется файл **models.py,** который применяется для определения моделей. Модель представляет **класс,** унаследованный от **django.db.models.Model**.

Изменим файл **models.py** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from django.db import models    class Person(models.Model):      name = models.CharField(max\_length=20)      age = models.IntegerField() |

1. Миграция

Каждая модель сопоставляется с определенной таблицей в базе данных. Однако пока у нас нет в бд таблицы, которая хранит объекты модели Person. И в этом случае нам надо создать и выполнить миграцию. Миграция преобразует базу данных в соответствии с определением моделей:

***python manage.py makemigrations***

В папке **migrations** мы обнаружим новый файл, который будет иметь примерно следующее содержимое:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | from django.db import migrations, models      class Migration(migrations.Migration):        initial = True        dependencies = [      ]        operations = [          migrations.CreateModel(              name='Person',              fields=[                  ('id', models.AutoField(auto\_created=True, primary\_key=True, serialize=False, verbose\_name='ID')),                  ('name', models.CharField(max\_length=20)),                  ('age', models.IntegerField()),              ],          ),      ] |

Это и есть миграция. Здесь можно заметить, что создается не два, а три поля - поле id, которое будет представлять первичный ключ, добавляется по умолчанию. Поэтому в принципе в самой модели нам не нужно явным образом определять какой-либо идентификатор.

Выполнить миграцию: ***python manage.py migrate***

Откроем базу данных **db.sqlite3**, которая есть в проекте, в специальной программе для просмотра БД SQLite, увидим, что она содержит ряд таблиц. Это служебные таблицы. Нас интересует таблица, которая называется по имени приложения и модели. Это таблица firstapp\_person, которая хранит данные модели Person.

1. Типы полей моделей

* **BinaryField()**: хранит бинарные данные
* **BooleanField()**: хранит значение True или False (0 или 1)
* **NullBooleanField()**: хранит значение True или False или Null
* **DateField()**: хранит дату
* **TimeField()**: хранит время
* **DateTimeField()**: хранит дату и время
* **DurationField()**: хранит период времени
* **AutoField()**: хранит целочисленное значение, которое автоматически инкрементируется, обычно применяется для первичных ключей
* **BigIntegerField()**: представляет число - значение типа Number, которое укладывается в диапазон от -9223372036854775808 до 9223372036854775807. В зависимости от выбранной СУБД диапазон может немного отличаться
* **DecimalField(decimal\_places=X, max\_digits=Y)**: представляет значение типа Number, которое имеет максимум X разрядов и Y знаков после запятой
* **FloatField()**: хранит, значение типа Number, которое представляет число с плавающей точкой
* **IntegerField()**: хранит значение типа Number, которое представляет целочисленное значение
* **PositiveIntegerField()**: хранит значение типа Number, которое представляет положительное целочисленное значение (от 0 до 2147483647)
* **PositiveSmallIntegerField()**: хранит значение типа Number, которое представляет небольшое положительное целочисленное значение (от 0 дролдо 32767)
* **SmallIntegerField()**: хранит значение типа Number, которое представляет небольшое целочисленное значение (от -32768 до 32767)
* **CharField(max\_length=N)**: хранит строку длиной не более N символов
* **TextField()**: хранит строку неопределенной длины
* **EmailField()**: хранит строку, которая представляет email-адрес. Значение автоматически валидируется встроенным валидатором EmailValidator
* **FileField()**: хранит строку, которая представляет имя файла
* **FilePathField()**: хранит строку, которая представляет путь к файлу длиной в 100 символов
* **ImageField()**: хранит строку, которая представляет данные об изображении
* **GenericIPAddressField()**: хранит строку, которая представляет IP-адрес в формате IP4v или IP6v
* **SlugField()**: хранит строку, которая может содержать только буквы в нижнем регистре, цифры, дефис и знак подчеркивания
* **URLField()**: хранит строку, которая представляет валидный URL-адрес
* **UUIDField()**: хранит строку, которая представляет UUID-идетификатор

1. CRUD. Операции с моделями

### Добавление данных

### Для добавления данных применяется метод **create()**:

|  |  |
| --- | --- |
| 1 | tom = Person.objects.create(name="Tom", age=23) |

Если добавление пройдет успешно, то объект будет иметь id, который можно получить через tom.id.

Однако в своей сути метод create() использует другой метод - **save()**, который мы также можем использовать отдельно для добавления объекта:

|  |  |
| --- | --- |
| 1  2 | tom = Person(name="Tom", age=23)  tom.save() |

После успешного добавления получим идентификатор добавленной записи с помощью  tom.id.

4.2 Получение из бд

Метод **get()** возвращает один объект по определенному условию, которое передается в качестве параметра:

|  |  |
| --- | --- |
| 1  2  3 | tom = Person.objects.get(name="Tom")    # получаем запись, где name="Tom"  bob = Person.objects.get(age=23)        # получаем запись, где age=23  tim = Person.objects.get(name="Tim", age=23)    # запись, где name="Tim" и age=23 |

При использовании этого метода надо учитывать, что он предназначен для выборки таких объектов, которые имеются в единичном числе в базе данных. Если в таблице не окажется подобного объекта, то мы получим ошибку имя\_модели.DoesNotExist. Если же в таблице будет несколько объектов, которые соответствуют условию, то будет сгенерированно исключение MultipleObjectsReturned. Поэтому следует применять данный метод с осторожностью.

Метод **get\_or\_create()** возвращает объект, а если его нет в бд, то добавляет в бд новый объект.

|  |  |
| --- | --- |
| 1  2  3 | bob, created = Person.objects.get\_or\_create(name="Bob", age=24)  print(bob.name)  print(bob.age) |

Метод возвращает добавленный объект (в данном случае переменная bob) и булевое значение (created), которое хранит True, если добавление прошло успешно.

Если необходимо получить все имеющиеся объекты, то применяется метод **all()**:

|  |  |
| --- | --- |
| 1 | people = Person.objects.all() |

Если надо получить все объекты, которые соответствуют определенному критерию, то применяется метод **filter()**, который в качестве параметра принимает критерий выборки:

|  |  |
| --- | --- |
| 1  2  3 | people = Person.objects.filter(age=23)  # использование нескольких критериев  people2 = Person.objects.filter(name="Tom", age=23) |

Метод **exclude()** позволяют исключить из выборки записи, которые соответвуют переданному в качестве параметра критерию:

|  |  |
| --- | --- |
| 1  2 | # исключаем пользователей, у которых age=23  people = Person.objects.exclude(age=23) |

Можно комбинировать два выше рассмотренных метода:

|  |  |
| --- | --- |
| 1  2 | # выбираем всех пользователей, у которых name="Tom" кроме тех, у которых age=23  people = Person.objects.filter(name="Tom").exclude(age=23) |

Метод **in\_bulk()** является более эффективным способом для чтения большого количества записей. Он возвращает словарь, то есть объект dict, тогда как методы all(), filter() и exclude() возвращают объект **QuerySet**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # получаем все объекты  people = Person.objects.in\_bulk()  for id in people:      print(people[id].name)      print(people[id].age)    # получаем объекты с id=1 и id=3  people2 = Person.objects.in\_bulk([1,3])  for id in people2:      print(people2[id].name)      print(people2[id].age) |

Метод in\_bulk возвращает словарь, где ключи представляют id объектов, а значения по этим ключам - собственно эти объекты, то есть в данном случае объекты Person.

### 4.3 Обновление

Для обновления объекта также применяется метод **save()**:

|  |  |
| --- | --- |
| 1  2  3 | bob = Person.objects.get(id=2)  bob.name = "Bob"  bob.save() |

В этом случае Django полностью обновляет объект, все его свойства, даже если мы их не изменяли. Чтобы указать, что нам надо обновить только определенные поля, следует использовать параметр update\_fields:

|  |  |
| --- | --- |
| 1  2  3 | bob = Person.objects.get(id=2)  bob.name = "Bobic"  bob.save(update\_fields=["name"]) |

Это позволит повысить производительность.

Другой способ обновления объектов представляет метод **update()** в сочетании с методом **filter**, которые вместе выполняют один запрос к базе данных:

|  |  |
| --- | --- |
| 1 | Person.objects.filter(id=2).update(name="Mike") |

Если нам не надо получать обновляемый объект, то данный способ позволит нам увеличить производительность взаимодействия с бд.

Иногда бывает необходимо изменить значение столбца в бд на основании уже имеющегося значения. В этом случае мы можем использовать функцию **F()**:

|  |  |
| --- | --- |
| 1  2  3 | from django.db.models import F    Person.objects.all(id=2).update(age = F("age") + 1) |

В данном случае полю age присваивается уже имеющееся значение, увеличенное на единицу.

При этом важно учитывать, что метод update обновляет все записи в таблице, которые соответствуют условию.

Если надо обновить вообще все записи, вне зависимости от условия, то необходимо комбинировать метод update с методом all():

|  |  |
| --- | --- |
| 1  2  3  4 | from django.db.models import F    Person.objects.all().update(name="Mike")  Person.objects.all().update(age = F("age") + 1) |

#### update\_or\_create()

Метод **update\_or\_create** обновляет запись, а если ее нет, то добавляет ее в таблицу:

|  |  |
| --- | --- |
| 1  2 | values\_for\_update={"name":"Bob", "age": 31}  bob, created = Person.objects.update\_or\_create(id=2, defaults = values\_for\_update) |

Метод update\_or\_create() принимает два параметра. Первый параметр представляет критерий выборки объектов, которые будут обновляться. Второй параметр представляет объект со значениями, которые будут переданы записям, которые соответствуют китерию из первого параметра. Если критерию не соответствует никаких записей, то в таблицу добавляется новый объект, а переменная created будет равна True.

4.4 Удаление

Для удаления мы можем вызвать метод **delete()** у удаляемого объекта:

|  |  |
| --- | --- |
| 1  2 | person = Person.objects.get(id=2)  person.delete() |

Если не требуется получение отдельного объекта из базы данных, тогда можно удалить объект с помощью комбинации методов filter() и delete():

|  |  |
| --- | --- |
| 1 | Person.objects.filter(id=4).delete() |

### 4.5 Просмотр строки запроса

С помо щью свойства **query** у результата запроса мы можем получить SQL-запрос, который выполнялся. Например:

|  |  |
| --- | --- |
| 1  2 | people = Person.objects.filter(name="Tom").exclude(age=34)  print(people.query) |

Данный код отобразит на консоли SQL-запрос типа следующего:

|  |  |
| --- | --- |
| 1  2 | SELECT "firstapp\_person"."id", "firstapp\_person"."name", "firstapp\_person"."age"  FROM "firstapp\_person" WHERE ("firstapp\_person"."name" = Tom AND NOT ("firstapp\_person"."age" = 34)) |

**Создание и получение объектов модели**

Рассмотрим создание и вывод объектов модели на примере. Возьмем проект из первой

темы данной главы.

В файле **models.py** определена модель Person:

|  |  |
| --- | --- |
| 1  2  3  4  5 | from django.db import models    class Person(models.Model):      name = models.CharField(max\_length=20)      age = models.IntegerField() |

В файле **views.py** пропишем два представления для получения данных из бд и для сохранения данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | from django.shortcuts import render  from django.http import HttpResponseRedirect  from .models import Person    # получение данных из бд  def index(request):      people = Person.objects.all()      return render(request, "index.html", {"people": people})    # сохранение данных в бд  def create(request):      if request.method == "POST":          tom = Person()          tom.name = request.POST.get("name")          tom.age = request.POST.get("age")          tom.save()      return HttpResponseRedirect("/") |

В функции index() получаем все данные с помощью метода Person.objects.all() и передаем их в шаблон index.html.

В функции create() получаем данные из запроса типа POST, сохраняем данные с помощью метода save() и выполняем переадресацию на корень веб-сайта (то есть на функцию index).

В папке templates определим шаблон **index.html**, который будет выводить данные на веб-страницу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>Модели в Django</title>  </head>  <body class="container">      <form method="POST" action="create/">          {% csrf\_token %}          <p>              <label>Введите имя</label><br>              <input type="text" name="name" />          </p>          <p>              <label>Введите возраст</label><br>              <input type="number" name="age" />          </p>          <input type="submit" value="Сохранить" >      </form>      {% if people.count > 0 %}      <h2>Список пользователей</h2>      <table>          <tr><th>Id</th><th>Имя</th><th>Возраст</th></tr>          {% for person in people %}          <tr><td>{{ person.id }}</td><td>{{ person.name }}</td><td>{{ person.age }}</td></tr>          {% endfor %}      </table>      {% endif %}  </body>  </html> |

В начале шаблона определена форма для добавления данных, которые потом будет получать функция create в POST-запросе. А ниже определена таблица, в которую выводятся данные из переданного из представления набора people.

И также в файле **urls.py** свяжем маршруты с представлениями:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | from django.urls import path  from firstapp import views    urlpatterns = [      path('', views.index),      path('create/', views.create),  ] |

Запустим проект и обратимся к приложению в браузере. Вначале добавим несколько объектов через форму на веб-странице:

И после каждого добавления мы увидим, как на веб-странице в таблице появляются новые данные: