# Lecture 4. Модели

## Создание моделей и миграции базы данных

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.

По умолчанию Django в качестве базы данных использует SQLite. Она очень проста в использовании и не требует запущенного сервера. Все файлы базы данных могут легко переноситься с одного компьютера на другой. Однако при необходимости мы можем использовать в Django большинство распространенных СУБД.

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

|  |
| --- |
| DATABASES = {  **'default'**: {  **'ENGINE'**: **'django.db.backends.sqlite3'**,  **'NAME'**: 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 файл db.sqlite3, который собственно и будет использоваться в качестве базы данных.

Чтобы использовать другие системы управления базами данных, необходимо будет установить соответствующий пакет.

|  |  |  |
| --- | --- | --- |
| СУБД | Пакет | Команда установки |
| PostgreSQL | psycopg2 | pip install psycopg2 |
| MySQL | mysql-python | pip install mysql-python |
| Oracle | cx\_Oracle | pip install cx\_Oracle |

**Создание моделей**

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.

Так, изменим файл models.py следующим образом:

|  |
| --- |
| **from** django.db **import** models   **class** Person(models.Model):  name = models.CharField(max\_length=20)  age = models.IntegerField() |

Здесь определена простейшая модель, которая называется Person и которая представляет человека. В модели определены два поля. Поле name представляет тип CharField - текстовое поле, которое хранит последовательность символов. Оно будет хранить имя человека. Для CharField обязательно надо указать параметр max\_length, который задает максимальную длину хранящейся строки. И поле age представляет тип IntegerField - числовое поле, которое хранит целые числа. Оно предназначено для хранения возраста человека.

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.

Вначале необходимо создать миграцию с помощью команды

|  |
| --- |
| python manage.py makemigrations |

После этого в приложении в папке migrations мы обнаружим новый файл, который будет иметь примерно следующее содержимое:

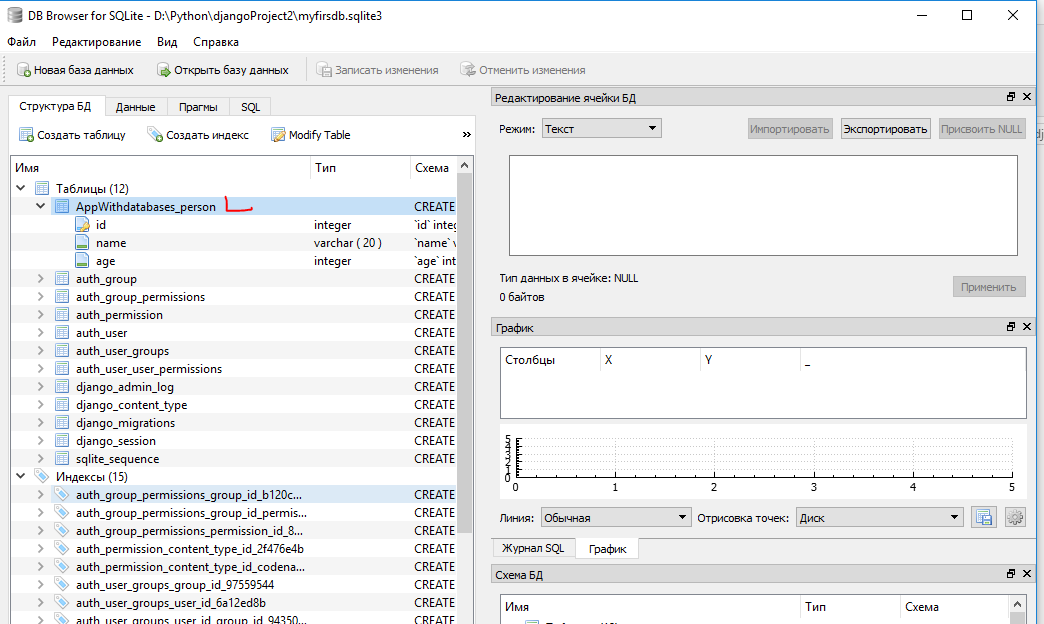
|  |
| --- |
| *# Generated by Django 4.0.4 on 2022-04-23 11:22* **from** django.db **import** migrations, models   **class** Migration(migrations.Migration):   initial = **True** dependencies = [  ]   operations = [  migrations.CreateModel(  name=**'Person'**,  fields=[  (**'id'**, models.BigAutoField(auto\_created=**True**, 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.

Теперь надо выполнить данную миграцию. Для этого выполняется команда

|  |
| --- |
| python manage.py migrate |

После этого, если мы откроем базу данных db.sqlite3, которая есть в проекте, в какой-нибудь специальной программе для просмотра БД SQLite, пример DB Browser for SQLite (<https://sqlitebrowser.org/dl/> ), то мы увидим, что она содержит ряд таблиц:



В основном это будут служебные таблицы. А нас прежде всего будет интересовать таблица, которая называется по имени приложения и модели. В моем случае это таблица AppWithdatabase\_person, которая и будет хранить данные модели Person.

## Типы полей моделей

Для определения моделей мы можем использовать следующие типы полей:

* 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 до 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 регитре, цифры, дефис и знак подчеркивания
* URLField(): хранит строку, которая представляет валидный URL-адрес
* UUIDField(): хранит строку, которая представляет universally unique identifier (UUID)-идетификатор
* Таблица сопоставления полей с типами в различных СУБД:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип | SQLite | MySQL | PostgreSQL | Oracle |
| BinaryField() | BLOB NOT NULL | longblob NOT NULL | bytea NOT NULL | BLOB NULL |
| BooleanField() | bool NOT NULL | bool NOT NULL | boolean NOT NULL | NUMBER(1) NOT NULL CHECK("Значение" IN(0,1)) |
| NullBooleanField() | bool NULL | bool NULL | boolean NULL | NUMBER(1) NOT NULL CHECK(("Значение" IN(0,1)) OR ("Значение" IS NULL)) |
| DateField() | date NULL | date NULL | date NULL | DATE NOT NULL |
| TimeField() | time NULL | time NULL | time NULL | TIMESTAMP NOT NULL |
| DateTimeField() | datetime NULL | datetime NULL | timestamp NULL | TIMESTAMP NOT NULL |
| DurationField() | bigint NOT NULL | bigint NOT NULL | interval NOT NULL | INTERVAL DAY(9) TO SECOND(6) NOT NULL |
| AutoField() | integer NOT NULL AUTOINCREMENT | integer AUTO\_INCREMENT NOT NULL | serial NOT NULL | NUMBER(11) NOT NULL |
| BigIntegerField | bigint NOT NULL | bigint NOT NULL | bigint NOT NULL | NUMBER(19) NOT NULL |
| DecimalField(decimal\_places=X, max\_digits=Y) | decimal NOT NULL | numeric(X, Y) NOT NULL | numeric(X, Y) NOT NULL | NUMBER(10, 3) NOT NULL |
| FloatField | real NOT NULL | double precision NOT NULL | double precision NOT NULL | DOUBLE PRECISION NOT NULL |
| IntegerField | integer NOT NULL | integer NOT NULL | integer NOT NULL | NUMBER(11) NOT NULL |
| PositiveIntegerField | integer unsigned NOT NULL | integer UNSIGNED NOT NULL | integer NOT NULL CHECK ("Значение" > 0) | NUMBER NOT NULL CHECK ("Значение" > 0) |
| PositiveSmallIntegerField | smallint unsigned NOT NULL | smallint UNSIGNED NOT NULL | smallint NOT NULL CHECK ("Значение" > 0) | NUMBER(11) NOT NULL CHECK ("Значение" > 0) |
| SmallIntegerField | smallint NOT NULL | smallint NOT NULL | smallint NOT NULL | NUMBER(11) NOT NULL |
| CharField(max\_length=N) | varchar(N) NOT NULL | varchar(N) NOT NULL | varchar(N) NOT NULL | NVARCHAR2(N) NULL |
| TextField() | text NOT NULL | longtext NOT NULL | text NOT NULL | NCLOB NULL |
| EmailField() | varchar(254) NOT NULL | varchar(254) NOT NULL | varchar(254) NOT NULL | NVARCHAR2(254) NULL |
| FileField() | varchar(100) NOT NULL | varchar(100) NOT NULL | varchar(100) NOT NULL | NVARCHAR2(100) NULL |
| FilePathField() | varchar(100) NOT NULL | varchar(100) NOT NULL | varchar(100) NOT NULL | NVARCHAR2(100) NULL |
| ImageField() | varchar(100) NOT NULL | varchar(100) NOT NULL | varchar(100) NOT NULL | NVARCHAR2(100) NULL |
| GenericIPAddressField() | char(39) NOT NULL | char(39) NOT NULL | inet NOT NULL | VARCHAR2(39) NULL |
| SlugField() | varchar(50) NOT NULL | varchar(50) NOT NULL | varchar(50) NOT NULL | NVARCHAR2(50) NULL |
| URLField() | varchar(200) NOT NULL | varchar(200) NOT NULL | varchar(200) NOT NULL | NVARCHAR2(200) NULL |
| UUIDField() | char(32) NOT NULL | char(32) NOT NULL | uuid NOT NULL | VARCHAR2(32) NULL |

## CRUD. Операции с моделями

При создании моделей они наследуют поведение от класса django.db.models.Model, который предоставляет ряд базовых операций с данными.

Рассмотрим ряд операций на примере модели Person:

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

**Добавление данных**

**create**

Для добавления данных применяется метод create():

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

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

**save**

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

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

После успешного добавления также можно получить идентификатор добавленной записи с помощью tom.id.

**Получение из бд**

**Получение одного объекта**

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

|  |
| --- |
| 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.

|  |
| --- |
| 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.

**all()**

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

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

**filter()**

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

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

**exclude()**

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

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

Можно комбинировать два выше рассмотренных метода:

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

**in\_bulk()**

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:

|  |
| --- |
| # 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.

**Обновление**

**save()**

Для обновления объекта также применяется метод save():

|  |
| --- |
| 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:

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

Это позволит повысить производительность.

**update()**

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

|  |
| --- |
| 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():

|  |
| --- |
| 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():

|  |
| --- |
| 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:

|  |
| --- |
| 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.

**Удаление**

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

|  |
| --- |
| 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():

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

**Просмотр строки запроса**

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

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

Данный код отобразит на консоли SQL-запрос типа следующего:

|  |
| --- |
| 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:

|  |
| --- |
| **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:

|  |
| --- |
| **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:

|  |
| --- |
| <!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:

|  |
| --- |
| **from** django.urls **import** path **from** AppWithdatabases **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:

## Редактирование и удаление объектов модели

Рассмотрим пример с редактированием и удалением объектов модели. Для этого продолжим работу с проектом из прошлой темы. Вначале добавим в файл views.py функции, которые будут собственно выполнять редактирование и удаление:

|  |
| --- |
| **from** django.shortcuts **import** render **from** django.http **import** HttpResponseRedirect **from** django.http **import** HttpResponseNotFound **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"**:  person = Person()  person.name = request.POST.get(**"name"**)  person.age = request.POST.get(**"age"**)  person.save()  **return** HttpResponseRedirect(**"/"**)   *# изменение данных в бд* **def** edit(request, id):  **try**:  person = Person.objects.get(id=id)   **if** request.method == **"POST"**:  person.name = request.POST.get(**"name"**)  person.age = request.POST.get(**"age"**)  person.save()  **return** HttpResponseRedirect(**"/"**)  **else**:  **return** render(request, **"edit.html"**, {**"person"**: person})  **except** Person.DoesNotExist:  **return** HttpResponseNotFound(**"<h2>Person not found</h2>"**)   *# удаление данных из бд* **def** delete(request, id):  **try**:  person = Person.objects.get(id=id)  person.delete()  **return** HttpResponseRedirect(**"/"**)  **except** Person.DoesNotExist:  **return** HttpResponseNotFound(**"<h2>Person not found</h2>"**) |

Первые две функции - create и index остаются те же самые, что и в прошлой теме.

Функция edit выполняет редактирование объекта. Функция в качестве параметра принимает идентификатор объекта в базе данных. И вначале по этому идентификатору мы пытаемся найти объект с помощью метода Person.objects.get(id=id). Поскольку в случае отсутствия объекта мы можем столкнуться с исключением Person.DoesNotExist, то соответственно нам надо обработать подобное исключение, если вдруг будет передан несуществующий идентификатор. И если объект не будет найден, то пользователю возващается ошибка 404 через вызов return HttpResponseNotFound().

Если объект найден, то обработка делится на две ветви. Если запрос POST, то есть если пользователь отправил новые изменненые данные для объекта, то сохраняем эти данные в бд и выполняем переадресацию на корень веб-сайта. Если запрос GET, то отображаем пользователю страницу edit.html с формой для редактирования объекта.

Функция delete аналогичным образом находит объет и выполняет его удаление.

Теперь добавим в папку templates файл edit.html со следующим содержимым:

|  |
| --- |
| <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="utf-8"**/>  <**title**>Модели в Django</**title**> </**head**> <**body class="container"**> <**form method="POST"**>  {% **csrf\_token** %}  <**p**>  <**label**>Введите имя</**label**><**br**>  <**input type="text" name="name" value="**{{ **person**.**name** }}**"**/>  </**p**>  <**p**>  <**label**>Введите возраст</**label**><**br**>  <**input type="number" name="age" value="**{{ **person**.**age** }}**"**/>  </**p**>  <**input type="submit" value="Сохранить"**> </**form**> </**body**> </**html**> |

Здесь определена форма для редактирования объекта. По нажатию на кнопку введенные на форму данные будут уходить по тому же адресу в запросе POST.

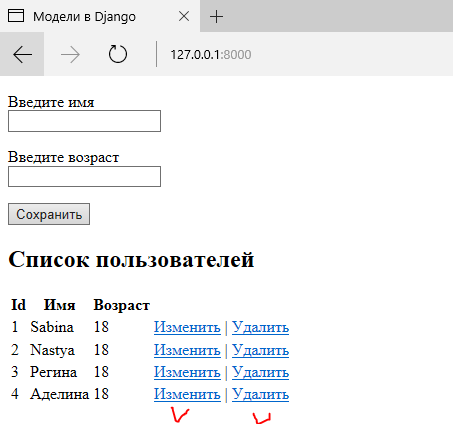
Чтобы не вводить вручную адреса для редактирования и удаления объектов изменим шаблон index.html, где выводится список объектов, добавив в него необходимые ссылки:

|  |
| --- |
| <!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**>  <**thead**>  <**th**>Id</**th**>  <**th**>Имя</**th**>  <**th**>Возраст</**th**>  <**th**></**th**>  </**thead**>  {% **for person in people** %}  <**tr**>  <**td**>{{ **person**.**id** }}</**td**>  <**td**>{{ **person**.**name** }}</**td**>  <**td**>{{ **person**.**age** }}</**td**>  <**td**><**a href="edit/**{{ **person**.**id** }}**"**>Изменить</**a**> | <**a href="delete/**{{ **person**.**id** }}**"**>Удалить</**a**></**td**>  </**tr**>  {% **endfor** %}  </**table**> {% **endif** %} </**body**> </**html**> |

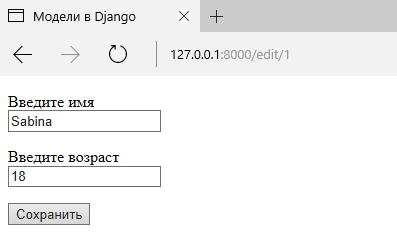
Далее в файле urls.py сопоставим функции edit и delete с маршрутами:

|  |
| --- |
| **from** django.urls **import** path **from** firstapp **import** views  urlpatterns = [  path(**''**, views.index),  path(**'create/'**, views.create),  path(**'edit/<int:id>/'**, views.edit),  path(**'delete/<int:id>/'**, views.delete), ] |

Запустим проект. На главной странице рядом с каждым объектом отобразятся две ссылки:



Нажав на кнопку редактирования, мы перейдем к форме, где мы сможем изменить значения выбранного объекта:



Соответственно нажав на кнопку удаления в таблице объектов, мы удалим выбранный объект.

## Отношение один ко многим (One to Many)

Рассмотрим организацию связи один ко многим, при которой одна главная сущность может быть связаны с несколькими зависимыми сущностями. Например, одна компания может выпускать несколько товаров:

|  |
| --- |
| **from** django.db **import** models   **class** Company(models.Model):  name = models.CharField(max\_length=30)   **class** Product(models.Model):  company = models.ForeignKey(Company, on\_delete=models.CASCADE)  name = models.CharField(max\_length=30)  price = models.IntegerField() |

В данном случае модель Company представляет производителя и является главной моделью, а модель Product представляет товар компании и является зависимой моделью.

Конструктор типа models.ForeignKey настраивает связь с главной сущностью. Первый параметр указывает, с какой моделью будет создаваться связь - в данном случае это модель Company. Второй параметр - on\_delete задает опцию удаления объекта текущей модели при удалении связанного объекта главной модели. Всего для параметра on\_delete мы можем использовать следующие значения:

* models.CASCADE: автоматически удаляет строку из зависимой таблицы, если удаляется связанная строка из главной таблицы
* models.PROTECT: блокирует удаление строки из главной таблицы, если с ней связаны какие-либо строки из зависимой таблицы
* models.SET\_NULL: устанавливает NULL при удалении связанной строка из главной таблицы
* models.SET\_DEFAULT: устанавливает значение по умолчанию для внешнео ключа в зависимой таблице. В этом случае для этого столбца должно быть задано значение по умолчанию
* models.DO\_NOTHING: при удалении связанной строки из главной таблицы не производится никаких действий в зависимой таблице

И в результате миграции в базе данных SQLite будут создаваться следующие таблицы:

|  |
| --- |
| CREATE TABLE `firstapp\_company` (      `id`    integer NOT NULL PRIMARY KEY AUTOINCREMENT,      `name`  varchar(30) NOT NULL  );  CREATE TABLE `firstapp\_product` (      `id`    integer NOT NULL PRIMARY KEY AUTOINCREMENT,      `name`  varchar(30) NOT NULL,      `price` integer NOT NULL,      `company\_id`    integer NOT NULL,      FOREIGN KEY(`company\_id`) REFERENCES `firstapp\_company`(`id`) DEFERRABLE INITIALLY DEFERRED  ); |

Чтобы показать в браузере изменим следующих файлов проекта:

views.py

|  |
| --- |
| **from** django.shortcuts **import** render **from** django.http **import** HttpResponseRedirect **from** django.http **import** HttpResponseNotFound **from** .models **import** Company   *# получение данных из бд* **def** index(request):  companies = Company.objects.all()  **return** render(request, **"index.html"**, context={**"companies"**: companies}) |

urls.py

|  |
| --- |
| **from** django.urls **import** path **from** AppWithdatabases **import** views  urlpatterns = [  path(**''**, views.index), ] |

Index.php

|  |
| --- |
| <!DOCTYPE **html**> <**html**> <**head**>  <**meta charset="utf-8"** />  <**title**>Hello Django</**title**> </**head**> <**body**>  <**ul**>  {% **for lan in companies** %}  <**li**>{{ **lan**.**name** }}</**li**>  {% **empty** %}  <**li**>Langs array is empty</**li**>  {% **endfor** %}  </**ul**> </**body**> </**html**> |

**Filter**

1. **Start width**

companies = Company.objects.filter(name\_\_startswith=**'Y'**)

1. Order by

companies = Company.objects.order\_by(**'name'**)

1. **Больше чем**

companies = Company.objects.filter(id\_\_gt=3)

1. **больше или равно** (>=)

companies = Company.objects.filter(id\_\_gte=3)

1. Меньше чем

companies = Company.objects.filter(id\_\_lt=3)

1. **Меньше или равно** (<=)

companies = Company.objects.filter(id\_\_lte=3)

1. Содержит

companies = Company.objects.filter(name\_\_contains=**'nd'**)

1. In

companies = Company.objects.filter(id\_\_in=[1,3,5])

В Django мы можем использовать объект Q() для реализации сложных SQL-запросов. Мы можем использовать объект Q() для представления оператора SQL, который можно использовать для некоторых операций с базой данных.

Этот оператор позволяет определять и повторно использовать условия, а также позволяет комбинировать их с такими операторами, как « ИЛИ » и « И ».

Объект Q() ( django.db.models.Q ) является контейнером для набора параметров ключевого слова. Эти параметры ключевых слов задаются так же, как «Поиски полей». Итак, давайте разберемся с его использованием на примере.

Теперь для демонстрации примера мы будем использовать ту же модель Company , что и в предыдущем разделе. И попробуем выбрать все объекты из модели Company, название которых начинается на «**Y**».

|  |
| --- |
| **from** django.shortcuts **import** render **from** django.http **import** HttpResponseRedirect **from** django.http **import** HttpResponseNotFound **from** .models **import** Company **from** django.db.models **import** Q  *# получение данных из бд* **def** index(request):  companies = Company.objects.filter(Q(name\_\_startswith=**"Y"**))  **return** render(request, **"index.html"**, context={**"companies"**: companies}) |

В приведенном выше примере сначала мы импортировали модель Company , а затем импортировали объект Q() из django.db.models . После этого мы создали QuerySet с методом фильтра, и в этом методе мы использовали объект Q() .

В Django мы не можем напрямую использовать оператор OR для фильтрации QuerySet. Для этой реализации мы должны использовать объект Q() . Используя объект Q() в методе фильтра, мы сможем использовать оператор ИЛИ между объектами Q() .

Давайте разберемся с реализацией на примере. В этом примере мы будем использовать модель Company. И попробуем выбрать из модели сотрудника все объекты, имя которых либо начинается на «Y», либо на «G».

|  |
| --- |
| **from** django.shortcuts **import** render **from** django.http **import** HttpResponseRedirect **from** django.http **import** HttpResponseNotFound **from** .models **import** Company **from** django.db.models **import** Q  *# получение данных из бд* **def** index(request):  companies = Company.objects.filter(Q(name\_\_startswith=**"Y"**) |Q(name\_\_startswith=**"G"**))  **return** render(request, **"index.html"**, context={**"companies"**: companies}) |

В примере мы используем метод filter в QuerySet. И в методе фильтра мы используем 2 объекта Q() с операторами поиска полей. И между этими операторами Q() мы определили оператор ИЛИ ( | )

В SQL мы используем ключевое слово DISTINCT в операторе SELECT для выбора уникальных записей. Обычно мы используем оператор SELECT DISTINCT , чтобы удалить все повторяющиеся записи из результата запроса.

Точно так же мы используем метод Different() в Django для удаления повторяющихся записей из QuerySet.

Примечание. Метод Different() с аргументом не поддерживается базой данных SQLite по умолчанию. Итак, в примере мы показали один из способов использования отдельного метода в базе данных SQLite.

|  |
| --- |
| **from** django.shortcuts **import** render **from** django.http **import** HttpResponseRedirect **from** django.http **import** HttpResponseNotFound **from** .models **import** Company **from** django.db.models **import** Q  *# получение данных из бд* **def** index(request):  companies = Company.objects.values(**'name'**).distinct()  **return** render(request, **"index.html"**, context={**"companies"**: companies}) |

**Операции с моделями**

Из определения таблиц мы видим, что Product связана с таблицей Company через столбец "company\_id". Однако в самом определении модели Product есть поле company, через которое можно получить связанную сущность:

|  |
| --- |
| *# получение id связанной с товаром компании* Product.objects.get(id=1).company.id  *# получение названия связанной с товаром компании* Product.objects.get(id=1).company.name  *# получение товаров, которые принадлежат к компании "Apple"* Product.objects.filter(company\_\_name=**"Apple"**) |

С помощью выражения модель\_\_свойство (два подчеркивания) можно использовать свойство главной модели для фильтрации по объектам зависимой модели.

Хотя с точки зрения модели Company она не имеет никаких свойств, которые бы связывали бы ее с моделью Product. Но с помощью синтаксиса "главная\_модель"."зависимая\_модель"\_set можно изменить направление связи.

|  |
| --- |
| **from** .models **import** Company, Product  apple = Company.objects.get(name=**"Apple"**)  *# получение всех товаров* apple.product\_set.all()  *# получение количества товаров* apple.product\_set.count()  *# получение товаров, название которых начинается на "iPhone"* apple.product\_set.filter(name\_\_startwith=**"iPhone"**) |

Причем с помощью выражения \_set можно выполнять операции добавления, изменения, удаления объектов зависимой модели из главной модели.

|  |
| --- |
| *# создаем объект Company* apple = Company.objects.create(name=**"Apple"**)  *# создание товара определенной компании* apple.product\_set.create(name=**"iPhone 8"**, price=67890)  *# отдельное создание объекта с последующим добавлением* ipad = Product(name=**"iPad"**, price=34560) *# при добавлении необходимо указать параметр bulk =False* apple.product\_set.add(ipad, bulk=**False**)  *# исключает из компании все товары,  # при этом товары остаются в бд, просто им не назначена компания # работает, если в зависимой модели ForeignKey(Company, null = True) # apple.product\_set.clear()  # то же самое, только в отношении одного объекта # ipad = Product.objects.get(name="iPad") # apple.product\_set.remove(ipad)* |

Стоит отметить три метода:

* add(): добавляет связь между объектом зависимой модели и объектом главной модели. В своей сути этот метод фактически вызывает для модели метод update() для добавления связи. Однако это требует, чтобы обе модели уже были в базе данных. И чтобы обойти это ограничение, применяется параметр bulk=False, для того, чтобы объект зависимой модели сразу был добавлен и для него была установлена связь.
* clear(): удаляет связь между всеми объектами зависимой модели и объектом главной модели. При этом сами объекты зависимой модели остаются в базе данных, и для их внешнего ключа устанавливается значение NULL. Поэтому данный метод будет работать, если в самой зависимой модели при установки связи использовался параметр null=True: ForeignKey(Company, null = True).
* remove(): также, как и clear() удаляет связь, только между одним объектом зависимой модели и объектом главной модели. При этом также все объекты остаются в бд. И также в самой зависимой модели при установки связи должен использоваться параметр null=True

## Отношение многие ко многим (Many to Many)

Связь многие ко многим описывает ситуацию, когда объект первой модели может одновременно ассоциироваться с несколькими объектами второй модели. И наоборот, один объект второй модели может также одновременно быть ассоциирован с несколькими объектами первой модели. Например, один студент может посещать несколько курсов, а один курс могут посещать несколько студентов.

Для создания отношения многие ко многим применяется тип ManyToManyField.

|  |
| --- |
| from django.db import models    class Course(models.Model):      name = models.CharField(max\_length=30)    class Student(models.Model):      name = models.CharField(max\_length=30)      courses = models.ManyToManyField(Course) |

В конструктор models.ManyToManyField передается сущность, с которой устанавливается отношение многие ко многим. В результате будет создаваться промежуточная таблица, через которую собственно и будет осуществляться связь.

В результате миграции в базе данных SQLite будут создаваться следующие таблицы:

|  |
| --- |
| CREATE TABLE `firstapp\_course` (      `id`    integer NOT NULL PRIMARY KEY AUTOINCREMENT,      `name`  varchar(30) NOT NULL  );  CREATE TABLE `firstapp\_student` (      `id`    integer NOT NULL PRIMARY KEY AUTOINCREMENT,      `name`  varchar(30) NOT NULL  );  CREATE TABLE `firstapp\_student\_courses` (      `id`    integer NOT NULL PRIMARY KEY AUTOINCREMENT,      `student\_id`    integer NOT NULL,      `course\_id` integer NOT NULL,      FOREIGN KEY(`student\_id`) REFERENCES `firstapp\_student`(`id`) DEFERRABLE INITIALLY DEFERRED,      FOREIGN KEY(`course\_id`) REFERENCES `firstapp\_course`(`id`) DEFERRABLE INITIALLY DEFERRED  ); |

В данном случае "firstapp\_student\_courses" выступает в качестве связующей таблицы. Она называется по шаблону имя\_таблицы + имя\_связующего\_поля\_из\_таблицы.

**Операции с моделями**

Через свойство courses в модели Student мы можем получать связанные со студентом курсы и управлять ими.

|  |
| --- |
| # создадим студента  tom = Student.objects.create(name="Tom")    # создадим один курс и добавим его в список курсов Тома  tom.courses.create(name="Algebra")    # получим все курсы студента  courses = Student.objects.get(name="Tom").courses.all()    # получаем всех студентов, которые посещают курс Алгебра  studes = Student.objects.filter(courses\_\_name="Algebra") |

Стоит отметить последний случай, где производится фильтрация студентов по посещаемому курсу. Для передачи в метод filter названия курса используется параметр, название которого начинается с названия свойства, через которое идет связь со второй моделью. И далее через два знака подчеркивания указывается имя свойства второй модели, например, courses\_\_name или courses\_\_id.

Однако в данном случае мы можем получить информацию о курсах студента через свойство courses, которое определено в модели Student. Однако что если мы хотим получить информацию о студентах по определенному курсу? В этом случае нам надо использовать синтаксис \_set.

|  |
| --- |
| # создадим курс  python = Course.objects.create(name="Python")    # создаем студента и добавляем его на курс  python.student\_set.create(name="Bob")    # отдельно создаем студента и добавляем его на курс  sam = Student(name="Sam")  sam.save()  python.student\_set.add(sam)    # получим всех студентов курса  students = python.student\_set.all()    # получим количество студентов по курсу  number = python.student\_set.count()    # удялем с курса одного студента  python.student\_set.remove(sam)    # удаляем всех студентов с курса  python.student\_set.clear() |

Стоит учитывать, что не всегда такая организация связи Многие ко Многим может подойти. Например, в данном случае создается промежуточная таблица, которая хранит id студента и id курса. Если нам надо в промежуточной таблице харнить еще какие-либо данные, например, дату зачисления студента на курс, его оценку и т.д., то такая конфигурация не подойдет. И в этом случае будет более оптимально создать промежуточную сущность вручную, которая связана отношением один ко многим с обеими моделями.

## Отношение один к одному (One to one)

Отношение один к одному предполагает, что одна строка из одной таблицы может быть связана только с одной сущностью из другой таблицы. Например, пользователь может иметь какие-либо данные, которые описывают его учетные данные. Всю базовую информацию о пользователе, типа имени, возраста, можно выделить в одну модель, а учетные данные - логин, пароль, время последнего входа в систему, количество неудачных входов и т.д. - в другую модель:

|  |
| --- |
| from django.db import models    class User(models.Model):      name = models.CharField(max\_length=20)    class Account(models.Model):      login = models.CharField(max\_length=20)      password = models.CharField(max\_length=20)      user = models.OneToOneField(User, on\_delete = models.CASCADE, primary\_key = True) |

Для создания этого отношения один к одному применяется конструктор типа models.OneToOneField(). Его первый параметр указывает, с какой моделью будет ассоциирована данная сущность (в данном случае ассоциация с моделью User). Второй параметр on\_delete = models.CASCADE говорит, что данные текущей модели (UserAccount) будут удаляться в случае удаления связанного объекта главной модели (User). Третий параметр primary\_key = True указывает, что внешний ключ (через который идет связь с главной моделью) в то же время будет выступать и в качестве первичного ключа. И соответственно создавать отдельное поле для первичного ключа не надо.

В результате миграции в базе данных SQLite будут создаваться следующие таблицы:

|  |
| --- |
| CREATE TABLE `firstapp\_user` (      `id`    integer NOT NULL PRIMARY KEY AUTOINCREMENT,      `name`  varchar(20) NOT NULL  );  CREATE TABLE `firstapp\_account` (      `login` varchar(20) NOT NULL,      `password`  varchar(20) NOT NULL,      `user\_id`   integer NOT NULL,      PRIMARY KEY(`user\_id`),      FOREIGN KEY(`user\_id`) REFERENCES `firstapp\_user`(`id`) DEFERRABLE INITIALLY DEFERRED  ); |

**Операции с моделями**

С помощью свойства users в модели Account мы можем манипулировать связанным объектом модели User:

|  |
| --- |
| # создадим пользователя  sam = User.objects.create(name="Sam")    # создадим аккаунт пользователя Sam  acc = Account.objects.create(login = "1234", password="6565", user=sam)    # изменяем имя пользователя  acc.user.name = "Bob"  # сохраняем изменения в бд  acc.user.save() |

Однако через модель User мы также можем оказывать влияние на связанный объект Account. Несмотря на то, что явным образом в модели User определено только одно свойство - name, при связи один к одному неявно создается еще одно свойство, которое называется по имени зависимой модели и которое указывает на связанный объект этой модели. То есть в данном случае это свойство будет называться "account":

|  |
| --- |
| # создадим пользователя  tom = User.objects.create(name="Tom")    # создадим аккаунт пользователя  acc = Account(login = "1234", password="6565")  tom.account = acc  tom.account.save()    # обновляем данные  tom.account.login = "qwerty"  tom.account.password = "123456"  tom.account.save() |