

ЛАБОРАТОРНАЯ РАБОТА № 7

Подключение базы данных к проекту

Цель: изучить принципы подключения и использования в проекте базы данных.

Содержание отчета: титульный лист, цель работы, задание, описание хода выполнения задания со скриншотами, листинги, вывод.

БД, SQL и ORM. Создание первой модели

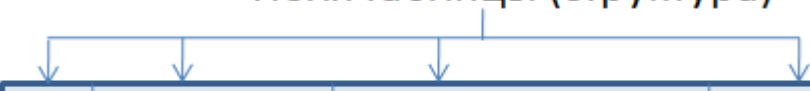
Рассмотрим следующий аспект паттерна MTV – модели. Модель отвечает за хранение и оперирование данными сайта. Django поддерживает следующие стандартные СУБД:

PostgreSQL, MariaDB, MySQL, Oracle и SQLite

<https://docs.djangoproject.com/en/4.2/ref/databases/>

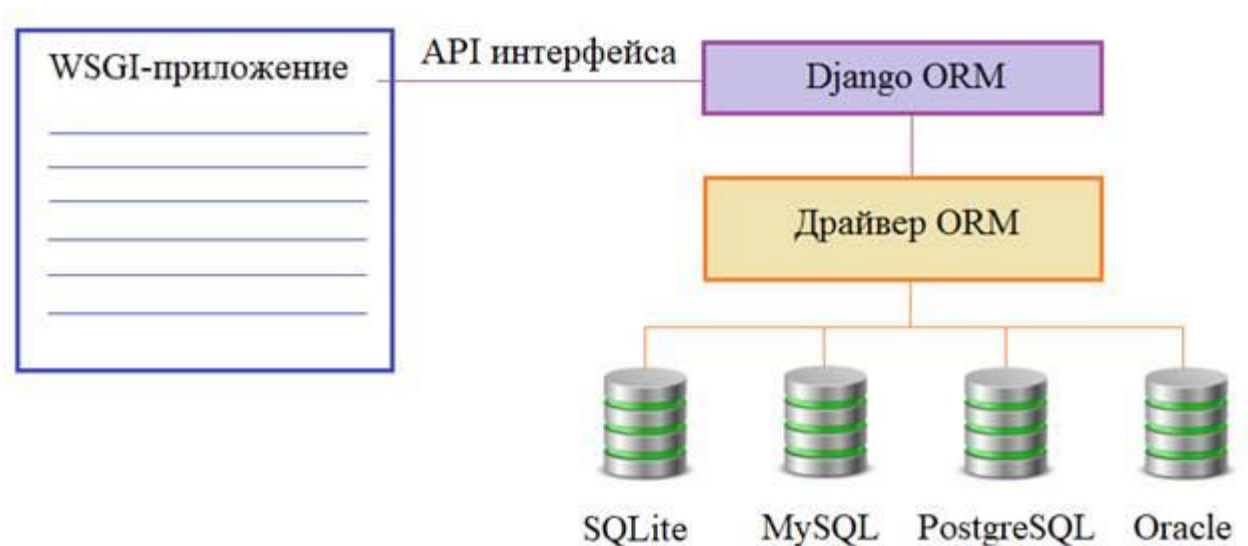
Все эти БД относятся к реляционным, то есть, позволяют хранить данные в виде связанных таблиц. Пример таблицы:

Поля таблицы (структура)



Записи	id	title	content	is_published
	1	А. Джоли	Биография А. Джоли	1
	2	М. Робби	Биография М. Робби	0
	3	Ума Турман	Биография У. Турман	1
	...			

Для создания таблиц, описания их структуры и наполнения данными используется язык SQL. Но чистый SQL здесь, как правило, не используется. Лишь в редких случаях требуется опускаться на этот нижний уровень взаимодействия с таблицами БД. Чтобы программист мог создавать универсальный программный код, не привязанный к конкретному типу БД, в Django встроен механизм взаимодействия с таблицами через объекты классов языка Python посредством технологии ORM (Object-Relational Mapping). Причем этот интерфейс универсален и на уровне WSGI-приложения не привязан к конкретной СУБД.



При работе с Django разработчику не нужно беспокоиться о подключении к БД и ее закрытию, когда пользователь покидает сайт. Фреймворк такие действия берет на себя и делает это весьма эффективно. Все что нам остается, это через модель взаимодействия выполнять команды API интерфейса, записывать, считывать и обновлять данные.

По умолчанию Django сконфигурирован для работы с БД SQLite. Текущую настройку БД можно посмотреть в файле **settings.py** пакета конфигурации:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Здесь словарь **DATABASES** по умолчанию определяет СУБД SQLite3 и путь к файлу БД **db.sqlite3**.

Подключение к другим СУБД происходит относительно просто в этом же словаре **DATABASES**, главное, иметь драйвер взаимодействия с ними. Для SQLite ничего дополнительно устанавливать не нужно. Но для работы с таблицами этой БД дополнительно будем использовать приложение DBSQLiteStudio, которое можно свободно скачать с официального сайта:

<https://sqlitestudio.pl>

Если мы откроем БД нашего сайта в этой программе, то увидим, что она пока не содержит ни одной таблицы. Давайте добавим первую модель, первый класс, который будет описывать таблицу **women** для хранения информации об известных женщинах. Структура этой таблицы будет такой:

women	
id:	Integer, primary key
title:	Varchar
content:	Text
time_create:	DateTime
time_update:	DateTime
is_published:	Boolean

Здесь первое поле **id** – это первичный ключ, принимающий уникальные числовые значения. Фактически, это идентификатор записи. Далее, идет поле **title** (заголовок статьи) в виде строки из определенного числа символов. Затем, поле **content**, представляющее собой текст статьи. Два следующих поля – это время создания статьи и время ее последнего изменения. Наконец, последнее поле **is_published** – это флаг публикации поста (**True** – опубликована; **False** – не опубликована).

Чтобы мы могли работать с такой таблицей, нам нужно объявить класс с этими полями. Этот класс в ORM называется моделью. Для его определения перейдем в файл **women/models.py**, в котором описываются модели текущего приложения, и здесь уже импортирован пакет **models**, содержащий базовый класс **Model**, на базе которого и строятся модели:

```
class Women(models.Model):
    title = models.CharField(max_length=255)
    content = models.TextField(blank=True)
    time_create =
models.DateTimeField(auto_now_add=True)
    time_update = models.DateTimeField(auto_now=True)
    is_published = models.BooleanField(default=True)
```

По умолчанию имя таблицы будет совпадать с именем класса, а ее структура определяться атрибутами, которые мы здесь определили. Обратите внимание, что нигде нет поля **id**. Но ошибки нет, такое поле создается автоматически у каждой таблицы. В действительности, оно уже прописано в базовом классе **Model** по всем правилам. Кроме того, атрибут **title** будет определять одноименное поле как текстовую строку с максимальным числом символов 255. Откуда известно, что для этого следовало использовать класс **CharField** с параметром **max_length**? Все это приведено в документации по фреймворку Django на следующей странице раздела ORM:

<https://docs.djangoproject.com/en/4.2/ref/models/fields/>

Здесь внушительный список самых разных классов, описывающих поля модели, и если щелкнуть по **CharField**, то увидим его описание и тот самый обязательный параметр **max_length**, что был использован для указания максимального числа символов в строке.

Итак, первые два поля (**id** и **title**) определены. Следующее поле **content** задано как текстовое с параметром **blank=True**. Данный параметр означает, что это поле может быть пустым, то есть, не содержать текста. Следующие два атрибута определены классом **DateTimeField**, предназначенным специально для работы со временем. У него есть два параметра:

- **auto_now_add** – позволяет фиксировать текущее время только в момент первого добавления записи в таблицу БД;
- **auto_now** – фиксирует текущее время всякий раз при изменении или добавлении записи в таблицу БД.

Эти параметры подходят для формирования времени атрибутов **time_create** и **time_update**.

Последний атрибут **is_published** определен через класс **BooleanField** с параметром **default=True**. Это означает, что по умолчанию значение поля в БД будет установлено в **True**, и статья будет считаться опубликованной.

Это лишь пример того, как можно описывать модель таблиц в БД. Причем, последовательность полей в таблице, по умолчанию, будет такой же как и в представленной модели.

Однако если сейчас запустить веб-сервер, то в нашей БД таблица создана не будет. Таблицы создаются отдельными командами, используя технику миграций.

Создание и запуск файлов миграций

Теперь нужно создать таблицу уже непосредственно в БД на основе сделанного нами описания. Для этого в Django существует механизм, известный как создание и выполнение миграций. Фактически, каждая миграция представляет собой отдельный файл (модуль) с текстом программы на языке Python, где описаны наборы команд на уровне ORM интерфейса, для создания таблиц определенных структур. При выполнении файла миграции в БД автоматически создаются новые или изменяются прежние таблицы, а также связи между ними. Каждый новый файл миграции помещается в папку **migrations** текущего приложения и описывает лишь изменения, которые произошли в структурах таблиц с прошлого раза. Их можно воспринимать как контролеры версий: всегда можно откатиться к предыдущей структуре и продолжить работу с прежней версией структур и связей между таблицами.

Давайте создадим миграцию. Для этого переходим в терминал и из корневого каталога проекта обращаемся к модулю **manage.py**, выполняя команду:

```
python manage.py makemigrations
```

Видим, что в каталоге **migrations** приложения **women** был успешно создан файл миграции с именем **0001_initial.py**. Если открыть этот файл, то увидим команду **CreateModel**, которая создает таблицу **women** с указанным набором полей, включая поле **id**.

Команду **makemigrations** следует выполнять каждый раз, когда у нас меняется хотя бы одна модель, иначе изменения не отразятся в файлах миграций и, как следствие, в самой БД. Сам веб-сервер этого не делает. Мы должны самостоятельно заранее подготовить (создать) нужные таблицы, а потом они просто используются для наполнения данными.

Чтобы посмотреть SQL-запрос, который будет выполнен при использовании данной миграции, можно записать следующую команду **sqlmigrate**:

```
python manage.py sqlmigrate women 0001
```

После этой команды мы указываем название приложения и порядковый номер миграции (только номер без **_initial.py**). Выполнив ее, в консоли увидим соответствующий SQL-запрос. Конечно, этот запрос может меняться в зависимости от выбранного типа БД.

Но пока мы лишь создали файл миграции. В БД по-прежнему нет никаких изменений. Чтобы изменения вступили в силу, необходимо выполнить файлы миграций. На самом деле их будет несколько, так как есть стандартные модели и для них также создаются файлы миграций, например, связанные с админ-панелью, авторизацией, сессиями и так далее, то есть с модулями, которые подключены по умолчанию к нашему проекту. Итак, для выполнения миграций запишем команду:

```
python manage.py migrate
```

Все выполнилось успешно, и, если открыть файл БД в SQLiteStudio, то увидим в ней множество таблиц, в том числе и нашу таблицу **women_women**. Ее имя было определено по имени приложения **Women** и имени модели, которое также **Women**. Если дважды щелкнуть по таблице, то увидим ее структуру, именно ту, что определили в нашей модели.

Вот так через механизм миграций создаются соответствующие таблицы в БД.

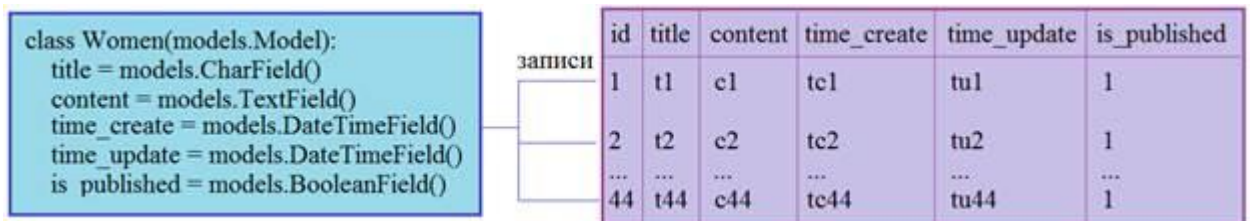
Понятие CRUD. Добавление записей в таблицу БД

Мы описали модель **Women** и создали на ее основе таблицу в БД, используя механизм миграций. Теперь научимся работать с этой таблицей, добавлять, выбирать, менять и удалять записи. Все эти операции сокращенно определяются аббревиатурой **CRUD** по первым буквам английских слов:

- **Create** – создание;
- **Read** – чтение;
- **Update** – изменение;
- **Delete** – удаление.

Используя ORM фреймворка Django, мы увидим, как выполняются данные команды в базовом исполнении. Почти все проекты, построенные на Django используют его встроенную ORM, не переходя на уровень SQL-запросов. В этом часто просто нет необходимости, так как ORM предоставляет богатые возможности по работе с БД. Кроме того, это обеспечивает независимость программного кода от конкретной используемой СУБД и если в будущем потребуется изменить тип БД, то сделать это будет предельно просто. Наконец, ORM в Django хорошо оптимизирует запросы по скорости выполнения и частоте обращения к таблицам БД, а также обеспечивает защиту от SQL-инъекций. Благодаря этому, даже начинающий веб-программист сможет создавать грамотный код по работе с БД.

Каждый экземпляр модели **Women** представляет собой одну отдельную запись таблицы **women_women** (далее просто **women**).



То есть, когда мы выбираем записи или создаем новые записи, то происходит работа именно с объектами класса **Women**. Сам же класс описывает структуру (набор и тип полей) таблицы.

Для демонстрации работы с ORM перейдем в консоль Django и в терминале выполним команду:

`python manage.py shell`

чтобы войти в консоль фреймворка. Первым делом выполним импорт модели:

```
from women.models import Women
```

Теперь, чтобы добавить новую запись в таблицу, нам достаточно создать экземпляр класса **Women** и передать в его конструктор значения именованных параметров, характеризующие поля таблицы:

```
Women(title='Анджелина Джоли', content='Биография  
Анджелины Джоли')
```

Мы видим, что объект был создан, но при этом были указаны только два аргумента. Дело в том, что атрибуты **time_create** и **time_update** у нас в модели инициализируются автоматически и определять их конкретными значениями нет необходимости. Поле **is_published** по умолчанию также принимает значение **True**. Поле **content** тоже можно было бы не определять (из-за параметра **blank=True**), но пусть оно содержит короткую строку.

Если сейчас перейти в SQLiteStudio и посмотреть содержимое таблицы **women_women**, то никаких записей не увидим. Дело в том, что модели в Django по умолчанию являются «ленивыми», создание экземпляра класса еще не означает добавление записи в таблицу. Это сделано специально. Мы можем в разных местах программы создавать объекты моделей и только в последний момент запускать их на исполнение, то есть, заносить информацию в БД. Благодаря этому Django имеет возможность оптимизировать SQL-запросы и излишне не нагружать СУБД.

Давайте укажем фреймворку сохранить созданную запись в таблице. Для этого нам понадобится какая-либо переменная, ссылающаяся на созданный объект, например, такая:

```
w1 = _
```

Здесь символ `'_'` – это специальная ссылка, в которой сохраняется результат последней операции. Если вывести переменную `w1`, то увидим строку:

<Women: Women object (None)>

Здесь **None** – это номер **id**, который принимает определенное, уникальное числовое значение в момент помещения записи в таблицу. Это делается с помощью метода `save()`:

```
w1.save()
```

и, выводя опять эту переменную в консоль, получаем строку:

<Women: Women object (1)>

Вместо **None** уже стоит значение **1**, то есть, запись была добавлена непосредственно в таблицу, и ей был присвоен идентификатор с номером **1**.

Перейдем в программу SQLiteStudio и убедимся, что данные в таблице действительно появились, причем остальные поля также были проинициализированы нужными значениями.

Непосредственно в программе, то есть, в консоли Django, мы можем оперировать всеми этими данными через ссылку **w1**:

```
w1.id # идентификатор
w1.title # заголовок
w1.time_create # время добавления записи
```

и так по всем полям (атрибутам класса). Помимо этих стандартных атрибутов объекты моделей содержат еще один часто используемый атрибут:

```
w1.pk # значение primary key
```

который совпадает с атрибутом **id**. Зачем было сделано такое дублирование? Дело в том, что поле **id** в таблицах имеет большое значение: часто именно по нему устанавливаются связи между таблицами. Поэтому по соглашению в Django решили определить атрибут со строго определенным именем **pk**, который будет всегда доступен и содержать номер текущей записи, либо значение **None**, если он не определен.

Конечно, непосредственное добавление записи в таблицу выполнялось с помощью SQL-запроса к БД. Чтобы его увидеть, нам нужно сначала импортировать модуль connection:

```
from django.db import connection
```

и обратиться к коллекции queries:

```
connection.queries
```

В консоли появится список словарей из выполненных запросов. У этого словаря имеются два ключа: **sql** – текст SQL-запроса; **time** – время выполнения этого запроса.

Создадим еще одну запись:

```
w2 = Women(title='Энн Хэтэуэй', content='Биография Энн Хэтэуэй')
```

Список **queries** остался прежним, так как запись еще не была добавлена в таблицу. Выполним команду:

```
w2.save()
```


Теперь в списке **queries** два запроса, а в таблице две записи. Эти два последних примера показывают, что объект класса `Women` можно наполнять информацией как угодно до момента непосредственной записи. Например, можно вначале создать экземпляр класса без аргументов:

```
w3 = Women()
```

а, затем, его локальным атрибутам присвоить требуемые значения:

```
w3.title = 'Джулия Робертс'  
w3.content = 'Биография Джулии Робертс'
```

После вызова метода `save`, запись будет добавлена в таблицу:

```
w3.save()
```

Установка улучшенной консоли `ipython`

Обратите внимание, что консоль фреймворка Django не очень удобна в работе, она даже не подсказывает нам возможные варианты команд, модулей и т.п. Но ее легко можно улучшить, установив специальный пакет **ipython**. Для этого в PyCharm перейдем в меню:

[File->Settings](#)

И в появившемся окне выберем **Project:sitewomen->Python Interpreter**. Нажмем на «плюс» и в строке поиска наберем **ipython**. Нажмем на кнопку **«Install Package»**. Пакет будет установлен для текущего выбранного интерпретатора.

Запустим снова консоль фреймворка Django командой:

`python manage.py shell`

и увидим несколько другое оформление. Если теперь начать вводить какую либо команду, например, «fr» и нажать на кнопку **Tab**, то увидим возможные варианты. Последующие нажатия на **Tab** будут подставлять соответствующие команды. Наберем:

```
from women.models import Women
```

Работать стало проще и удобнее. Выйдем из консоли с помощью команды **exit**.

Установка пакета **django-extensions**

Установим еще один пакет, который может упростить отработку ORM-команд в консоли. Он называется **django-extensions** и устанавливается аналогично предыдущему пакету:

<https://github.com/django-extensions/django-extensions>

После установки мы должны его прописать в коллекции **INSTALLED_APPS** файла конфигурации **settings.py**. Сделаем это:

```
INSTALLED_APPS = [  
    ...  
    'django_extensions',  
    'women.apps.WomenConfig',  
]
```

После этого, нам становится доступна новая команда **shell_plus**, которую выполним с ключом **--print-sql**:

`python manage.py shell_plus --print-sql`

Запустится новая оболочка с поддержкой ранее установленного **ipython** и с автоматической распечаткой выполняемых SQL-запросов. Давайте в этом убедимся. Создадим объект класса **Women** (в оболочке **shell_plus** модели импортируются автоматически):

```
a = Women(title="Екатерина Гусева", content="Биография  
Екатерины Гусевой")
```

и вызовем метод **save()**:

```
a.save()
```

В результате нам автоматически выводится SQL-запрос, который добавил эту запись в БД.

Методы выбора записей из таблиц

Перейдем в консоль **shell_plus**:

`python manage.py shell_plus --print-sql`

Каждый класс модели содержит специальный статический объект **objects**, который наследуется от базового класса **Model** и представляет собой ссылку на специальный класс **Manager**. В этом легко убедиться, если выполнить строчку:

```
Women.objects
```

В консоли увидим:

```
<django.db.models.manager.Manager object at 0x0399CA00>
```

Этот объект **objects** еще называют **менеджером записей**, и у него есть несколько полезных методов. Начнем с метода **create()** создания новой записи в таблице:

```
w = Women.objects.create(title='Ума Турман',  
content='Биография Ума Турман')
```

Если теперь обратиться к свойству:

```
w.pk
```

то увидим значение 5, то есть запись была автоматически добавлена в БД. Нам здесь не нужно отдельно вызывать метод **save()**, все происходит «на лету». Добавим еще одну запись, не присваивая результат какой-либо переменной:

```
Women.objects.create(title='Кира Найтли',  
content='Биография Киры Найтли')
```

В таблице появилась еще одна запись с идентификатором 6.

Выборка записей из таблицы методом **all()**

Как прочитать данные из таблицы **women**? Для этого воспользуемся встроенным менеджером записей – объектом **objects** и выполним метод **all()**:

```
Women.objects.all()
```

На выходе получаем список **QuerySet**:

```
<QuerySet [<Women: Women object (1)>, <Women: Women object (2)>,  
<Women: Women object (3)>, <Women: Women object (4)>, <Women: Women  
object (5)>, <Women: Women object (6)>]>
```

Но такая информация, когда показываются записи только с идентификаторами, не очень информативна. Давайте вместо идентификаторов будем выводить заголовок – поле **title**. Для этого в модели достаточно переопределить магический метод **__str__**:

```
class Women(models.Model):  
...  
    def __str__(self):
```

```
return self.title
```

Чтобы изменения вступили в силу, выйдем из консоли Django (команда **exit**) и снова зайдем в **shell_plus**. Импортируем модель:

```
from women.models import Women
```

И выполняем команду выбора всех записей:

```
Women.objects.all()
```

Теперь в консоли отображаются заголовки записей, а не их **id**:

```
<QuerySet [<Women: Анджелина Джоли>, <Women: Энн Хэтэуэй>, <Women: Джулия Робертс>, <Women: Екатерина Гусева>, <Women: Ума Турман>, <Women: Кира Найтли>]>
```

Получить отдельную запись из этого списка можно по индексу, например:

```
w = Women.objects.all()[0]
```

Причем, **w** – это ссылка на объект класса **Women**, у которого имеются указанные нами атрибуты. Эти атрибуты содержат информацию о текущей записи:

```
w.title  
w.content
```

Или же можно указать срез, например:

```
w = Women.objects.all()[:3]
```

Обратите внимание, на данный момент ORM не делает обращения к БД, т.к. мы не обращаемся к записям в консоли. Но если отобразить переменную **w**:

```
w
```

то увидим SQL-запрос и отображение коллекции **QuerySet** для первых трех записей.

Также список записей можно получить и затем перебрать циклом **for**, например, так:

```
ws = Women.objects.all()  
for w in ws:  
    print(w)
```

Выборка записей по фильтру (критерию)

Обычно в программе нам требуется не все, а несколько записей, выбранных по какому-либо критерию (условию). Для этого вместо метода **all()** следует использовать метод **filter()**, например, так:

```
Women.objects.filter(title='Энн Хэтэуэй')
```

На выходе получим одну запись, у которой **id** равен 2. Сам же SQL-запрос, выглядит следующим образом:

```
'SELECT "women_women"."id", "women_women"."title",  
"women_women"."content", "women_women"."photo",  
"women_women"."time_create", "women_women"."time_update",  
"women_women"."is_published" FROM "women_women" WHERE  
"women_women"."title" = \'Энн Хэтэуэй\' LIMIT 21'
```

То есть, метод **filter** добавляет ключевое слово **WHERE** для формирования выборки по условию. Причем, если условию не будет удовлетворять ни одна запись, то получим пустой список:

```
Women.objects.filter(title='Энн')
```

Этот пример также показывает, что ищется строка целиком, а не ее часть.

Если мы хотим сделать выборку по записям, у которых **id** больше или равен 2, то прописать условие в виде:

```
Women.objects.filter(pk > 2)
```

нельзя, так как **pk** — это именованный параметр и ему нужно явно присваивать определенное значение. Поэтому в Django атрибутам, определенным в модели, можно дополнительно прописывать следующие lookup'ы:

- <имя атрибута>__gte — сравнение больше или равно (>=);
- <имя атрибута>__gt — сравнение больше (>);
- <имя атрибута>__lte — сравнение меньше или равно (<=);
- <имя атрибута>__lt — сравнение меньше (<).

Подробную информацию о них можно посмотреть в документации:

<https://docs.djangoproject.com/en/4.2/ref/models/queries/#field-lookups>

В результате, искомый фильтр можно записать следующим образом:

```
Women.objects.filter(pk__gte=2)
```

Рассмотрим еще несколько примеров lookups. Если мы запишем команду в виде:

```
Women.objects.filter(title='ли')
```

то будут выбираться все записи, у которых заголовки точно соответствует строке «ли». Если же нам нужно выбрать те, у которых в заголовке присутствует этот фрагмент, то следует воспользоваться фильтром **contains**. Он позволяет находить строки по их фрагменту, учитывая регистр букв. Например, команда:

```
Women.objects.filter(title__contains='ли')
```

выдаст список всех женщин, в заголовке у которых присутствует фрагмент «ли». На уровне SQL-запроса это делается с помощью фрагмента:

«WHERE title LIKE '%ли%'»

Похожий фильтр **icontains** осуществляет поиск без учета регистра символов. Однако, если мы запишем вот такую команду:

```
Women.objects.filter(title__icontains='ЛИ')
```

то получим пустой список. Дело в том, что СУБД SQLite не поддерживает регистронезависимый поиск для русских символов (и, вообще, для всех не ASCII-символов), поэтому получаем пустой список. Другие СУБД, как правило, отрабатывают все это корректно. В случае с латинскими символами в SQLite поиск всегда проходит как регистронезависимый.

Следующий фильтр **in** позволяет указывать через список выбираемые записи по значениям. Например, выберем записи с **id** равными 2, 5, 11, 12:

```
Women.objects.filter(pk__in=[2,5,11,12])
```

Если по условию нужно отработать сразу несколько фильтров, то они указываются через запятую:

```
Women.objects.filter(pk__in=[2,5,11,12],  
is_published=1)
```

При этом на уровне SQL-запросов формируется связка через логическое И:

WHERE ("women_women"."is_published" AND "women_women"."id" IN (2, 5,
11, 12))

то есть запись выбирается, если оба критерия срабатывают одновременно. Чтобы определять условия через логическое ИЛИ используется специальный

класс **Q**, он будет рассмотрен позднее. По аналогии используются и все остальные фильтры фреймворка Django.

Противоположным по действию является метод **exclude()**. Он выбирает записи, не удовлетворяющие указанному условию. Например:

```
Women.objects.exclude(pk=2)
```

Будут выбраны все записи, кроме записи с id равным 2.

Если нам нужно выбрать только одну запись (обычно используя ее идентификационный номер **id**), то можно использовать метод **get()**:

```
Women.objects.get(pk=2)
```

Получим саму запись (без списка). Если записей, удовлетворяющих условию будет несколько:

```
Women.objects.get(pk__gte=2)
```

или они не будут существовать:

```
Women.objects.get(pk=20)
```

то метод **get()** генерирует исключения.

Сортировка записей

Снова перейдем в консоль **shell_plus**:

```
python manage.py shell_plus --print-sql
```

Для сортировки записей по указанному полю используется метод **order_by()**, например, следующим образом:

```
Women.objects.all().order_by('title')
```

Получаем SQL-запрос, в котором появляется ключевое слово **ORDER BY** с флагом **ASC** (сортировка по возрастанию). На выходе получим список:

```
<QuerySet [<Women: Анджелина Джоли>, <Women: Джулия Робертс>,  
<Women: Екатерина Гусева>, <Women: Кира Найтли>, <Women: Ума  
Турман>, <Women: Энн Хэтэуэй>]>
```

Ту же самую команду можно записать в виде:

```
Women.objects.order_by('title')
```


Результат будет абсолютно тем же самым.

Вообще, метод `order_by()` можно применять к любой коллекции `QuerySet`. Например, с помощью фильтра отобрать несколько записей, а затем, выполнить их фильтрацию:

```
Women.objects.filter(pk__lte=4).order_by('title')
```

Здесь отбираются все записи, у которых `id` меньше или равен 4 и сортируются по полю `title` в порядке возрастания (используется лексикографическое сравнение строк). Этот пример показывает, как методы можно цепочкой выполнять друг за другом: сначала выбрали записи по условию, а затем их отсортировали. Точно также по цепочке можно выполнять многие другие методы ORM.

Если нам нужно изменить порядок сортировки на противоположный, то перед именем поля достаточно поставить знак минус:

```
Women.objects.order_by('-time_update')
```

В этом случае в SQL-запросе после ключевого слова **ORDER BY** появляется флаг **DESC**, означающий сортировку по убыванию.

Вложенный класс Meta

При необходимости, в любой модели мы можем определить некоторые глобальные настройки. Например, по умолчанию выполнять сортировку статей по убыванию времени их создания. Для этого внутри класса модели следует прописать еще один класс **Meta** и в нем определить атрибут `ordering` следующим образом:

```
class Women(models.Model):
    ...
    class Meta:
        ordering = ['-time_create']
        indexes = [
            models.Index(fields=['-time_create']),
        ]

    def __str__(self):
        return self.title
```

И дополнительно здесь сразу указано, что это поле должно быть индексируемым, чтобы сортировка выполнялась быстрее. Вообще у этого класса множество разных атрибутов. Подробнее о них можно посмотреть на следующей странице документации:

<https://docs.djangoproject.com/en/4.2/ref/models/options/>

Если теперь снова зайти в консоль **shell_plus**:

```
python manage.py shell_plus --print-sql
```

то при извлечении всех записей:

```
Women.objects.all()
```

они будут следовать в обратном порядке согласно атрибуту **ordering** класса **Meta**.

Изменение записей

В самом простом случае, для изменения какой-либо записи, ее можно сначала прочитать из БД, например, с помощью метода **get()**:

```
wu = Women.objects.get(pk=2)
```

Затем, присвоить атрибутам объекта **Women** другие значения, например:

```
wu.title = 'Марго Робби'  
wu.content = 'Биография Марго Робби'
```

Сохраняем новые данные:

```
wu.save()
```

и в таблице видим, что вторая запись содержит новую, измененную информацию. При этом последний SQL-запрос имеет следующий вид:

```
'UPDATE "women_women" SET "title" = \'Марго Робби\', "content" =  
\'Биография Марго Робби\', "photo" = \'\', "time_create" = \'2021-01-03  
09:27:56.511898\', "time_update" = \'2021-01-03 11:57:06.800768\',  
"is_published" = 1 WHERE "women_women"."id" = 2'
```

Это первый подход к изменению записей, когда у нас уже имеется объект, который нужно изменить. Но, если нам нужно, например, у всех записей поле **is_published** установить в ноль, то можно использовать метод **update()**:

```
Women.objects.update(is_published=0)
```

Если мы хотим поменять значение поля **is_published** только для первых четырех записей, например, установить его в единицу, то, сделав это через срезы:

```
Women.objects.all()[:4].update(is_published=1)
```

получим ошибку. Метод **update()** нельзя комбинировать со срезами. Но после метода **filter()** его можно вызывать. Поэтому следующая команда даст нужный нам результат:

```
Women.objects.filter(pk__lte=4).update(is_published=1)
```

Также получим ошибку, если метод **update()** вызвать для объекта класса **Women**, например, так:

```
Women.objects.get(pk=5).update(is_published=1)
```

И это понятно, так как для изменения записей в SQL-запросах на первом месте должно стоять ключевое слово **UPDATE**, а при формировании выборки – ключевое слово **SELECT**. Это два разных типа запросов и комбинировать их нельзя. Отсюда и получаем ошибку их выполнения.

Удаление записей

Последняя базовая операция – удаление записей, выполняется аналогично изменению. Сначала нужно выбрать те записи, что мы собираемся удалить, например, вот так:

```
wd = Women.objects.filter(pk__gte=5)
```

А, затем, выполняем для них метод **delete()**:

```
wd.delete()
```

Записи с **id** равным 5 и 6 были удалены из таблицы **women**.

Мы рассмотрели некоторые возможности ORM Django. Более детальную информацию можно почитать на странице документации:

docs.djangoproject.com/en/4.2/topics/db/queries/

Слагги в URL-адресах

Слаг (slug) — часть URL-адреса, которая идентифицирует определенную страницу. Слаг обычно состоит из набора маленьких латинских букв, цифр, символов подчеркивания и дефиса. Например, «samsung-galaxy-s21» — это слаг в адресе <https://example.com/shop/smartphones/samsung-galaxy-s21>. Использование слогов – рекомендуемая практика в веб-программировании. Такие страницы лучше ранжируются поисковыми системами и понятнее конечному пользователю.

Сделаем отображение отдельных статей по их слагу. Для этого сначала сделаем отображение статей по их идентификатору, а затем, заменим адрес

на слог. У нас уже есть функция-заглушка **show_post()** в файле **women/views.py**. Мы ее перепишем, следующим образом:

```
def show_post(request, post_id):
    post = get_object_or_404(Women, pk=post_id)

    data = {
        'title': post.title,
        'menu': menu,
        'post': post,
        'cat_selected': 1,
    }

    return render(request, 'women/post.html',
context=data)
```

Здесь функция **get_object_or_404** выбирает одну запись из таблицы **Women**, которая имеет идентификатор, равный **post_id**, либо генерирует исключение 404, если запись не была найдена. Это аналог следующего кода:

```
try:
    post = Women.objects.get(pk=post_id)
except Women.DoesNotExist:
    raise Http404("Page Not Found")
```

Но прописывать постоянно такие строчки не очень удобно, поэтому в Django для таких случаев заготовлена специальная функция **get_object_or_404()**.

Далее, формируется словарь из параметров шаблона и отображается страница на основе шаблона **post.html**. У нас пока нет такого файла, добавим его со следующим содержимым:

```
{% extends 'base.html' %}

{% block content %}
<h1>{{post.title}}</h1>

{% if post.photo %}
<p ></p>
{% endif %}

{{post.content|linebreaks}}
{% endblock %}
```

Здесь все достаточно очевидно. Отображаем заголовок **h1**, затем фотографию статьи, если она есть, и потом уже содержимое самой статьи.

Если теперь перейти по ссылке, то увидим статью, взятую по указанному индексу:

<http://127.0.0.1:8000/post/1/>

Если же указать неверный адрес, то получим исключение 404. Исключения в таком развернутом виде отображаются только в режиме отладки сайта. При эксплуатации с константой **DEBUG = False** вместо исключения отображается заготовленная страница 404.

Добавление слага

Следующим шагом сделаем отображение статей по их слаг. Для этого в модели **Women** необходимо прописать еще одно поле, которое так и назовем – **slug**:

```
class Women(models.Model):
    title = models.CharField(max_length=255,
        verbose_name="Заголовок")
    slug = models.SlugField(max_length=255,
        unique=True, db_index=True, verbose_name="URL")
    ...
```

Мы его определили после поля **title**, указав уникальным и индексируемым. Однако, если сейчас попытаться создать миграцию для внесения этих изменений в структуру таблицы **women**:

[python manage.py makemigrations](#)

увидим предупреждение, что поле не может быть пустым (так как у нас есть записи в таблице). Чтобы таблица была сформирована, временно пропишем еще два параметра **blank=True** и **default=''**, а **unique=True** уберем:

```
slug = models.SlugField(max_length=255, db_index=True,
    blank=True, default='')
```

Снова выполним команду:

[python manage.py makemigrations](#)

и видим, что теперь никаких ошибок нет, и был создан еще один файл миграций со значением 2.

Применим эти миграции для добавления нового поля в таблицу **women**:

`python manage.py migrate`

Видим, что в таблице было успешно создано новое поле **slug** и размещено в самом конце.

Заполним теперь его уникальными значениями. Для этого перейдем в терминал:

`python manage.py shell_plus`

и выполним следующие команды:

```
for w in Women.objects.all():
    w.slug = 'slug-'+str(w.pk)
    w.save()
```

Теперь у каждой записи свой слаг в виде строки «slug-<идентификатор>».

Вернемся к определению модели. Изменим поле **slug** следующим образом:

```
slug = models.SlugField(max_length=255, db_index=True,
unique=True)
```

Создадим миграцию для внесения изменений в таблицу:

`python manage.py makemigrations`

и применим ее:

`python manage.py migrate`

Теперь поле **slug** у нас должно быть уникальным, причем это условие уровня базы данных, то есть СУБД дополнительно будет проверять на уникальность поля **slug** каждой добавляемой или изменяемой записи.

База данных готова, и теперь можно сделать отображение статей по слагу. Для этого откроем файл **women/urls.py** и в списке **urlpatterns** изменим маршрут для постов на следующий:

```
path('post/<slug:post_slug>/', views.show_post,
name='post'),
```

Затем, в файле **women/views.py** немного поменяем функцию представления **show_post**:

```
def show_post(request, post_slug):
```

```
post = get_object_or_404(Women, slug=post_slug)
...
```

Теперь, если перейти по адресу:

<http://127.0.0.1:8000/post/slug-1/>

то увидим отображение первой статьи по слаг, а не идентификатору статьи, что несколько понятнее для человека. Но на данный момент у нас слаг не очень понятный, поэтому перейдем в таблицу **women** и у всех статей вручную поменяем слаг на более удобочитаемые:

- andzhelina-dzholi
- margo-robby
- dzhuliya-roberts
- ekaterina-guseva

Сохраняем изменения, переходим по адресу:

<http://127.0.0.1:8000/post/andzhelina-dzholi/>

и видим пост по Анджелине Джоли.

Изменение ссылок в шаблоне. Метод `get_absolute_url()`

Однако, если мы сейчас перейдем на главную страницу сайта, то ссылки кнопки «Читать пост» у нас по-прежнему отображаются с идентификатором. Исправим это и сделаем ссылки со слагами. Для этого в модели **Women** (в файле **women/models.py**) будем формировать нужный нам URL-адрес по параметру **slug** с помощью метода **get_absolute_url()** следующим образом:

```
class Women(models.Model):
...
    def get_absolute_url(self):
        return reverse('post', kwargs={'post_slug':
self.slug})
...
```

В шаблоне **index.html** вызовем этот метод для каждого объекта класса **Women**:

```
{% extends 'base.html' %}

{% block content %}
<ul class="list-articles">
    {% for p in posts %}
```



```

        {% if p.is_published %}
            <li><h2>{{p.title}}</h2>
        {% autoescape off %}
        {{p.content|linebreaks|truncatewords:50}}
        {% endautoescape %}
            <div class="clear"></div>
            <p class="link-read-
post"><a href="{{ p.get_absolute_url }}">Читать
пост</a></p>
            </li>

        {% endif %}
        {% endfor %}
</ul>
{% endblock %}

```

Вспоминим, что **p** в цикле **for** как раз ссылается на объекты класса **Women**, у которого теперь есть атрибут **get_absolute_url**. Обратите внимание, что при указании этого метода, мы не пишем в конце круглые скобки, т.к. он здесь самостоятельно не вызывается. Вызов сделает функция **render** при обработке этого шаблона.

Почему мы заменили тег **url** методом **get_absolute_url**? Представьте, что в будущем шаблон этой ссылки снова изменился для вывода постов по **id**. Тогда, при использовании тега **url**, нам пришлось бы менять эти ссылки в каждом шаблоне, заменяя **self.slug** на **self.pk**. В этом как раз неудобство и источник потенциальных ошибок. А благодаря определению нового метода **get_absolute_url()** нам достаточно изменить маршрут только в нем и это автоматически скажется на всех шаблонах, где используется его вызов.

Второй важный момент функции **get_absolute_url()** заключается в том, что, согласно конвенции, модули Django используют этот метод в своей работе (если он определен в модели). Например, стандартная админ-панель обращается к этому методу для построения ссылок на каждую запись наших моделей.

В свою очередь, тег **{% url %}** имеет смысл применять для построения ссылок не связанных с моделями или, для ссылок без параметров, используя только имена маршрутов.

В функции представления **index()** укажем читать все посты из БД, у которых флаг **is_published** равен 1, и передавать их шаблон:

```

def index(request):
    posts = Women.objects.filter(is_published=1)
    data = {
        'title': 'Главная страница',

```

```

        'menu': menu,
        'posts': posts,
    }

    return render(request, 'women/index.html',
context=data)

```

Обновляем главную страницу сайта и видим, что теперь посты доступны по слаггу, а не идентификатору.

Создание пользовательского менеджера модели

В функции представления **index** мы читали все опубликованные посты из БД и отображали их в виде списка на главной странице. При этом использовался менеджер **objects** модели **Women**. Но фреймворк Django позволяет нам в моделях создавать свои собственные менеджеры. Определим такой менеджер, который будет возвращать только опубликованные статьи.

Перейдем в файл **models.py** приложения **women** и объявим новый класс, который будет описывать менеджер для моделей, следующим образом:

```

class PublishedModel(models.Manager):
    def get_queryset(self):
        return
super().get_queryset().filter(is_published=1)

```

Мы здесь объявляем метод **get_queryset()**, который вызывается для получения списка записей из таблиц БД с помощью этого менеджера. Соответственно, в самом методе идет обращение к аналогичному методу базового класса, но с отбором только опубликованных записей.

Затем, в классе модели **Women** необходимо создать объект этого класса менеджера, например, так:

```

class Women(models.Model):
    ...
    objects = models.Manager()
    published = PublishedModel()

    def get_absolute_url(self):
        return reverse('post', kwargs={'post_slug':
self.slug})

    def __str__(self):
        return self.title

```

Обратите внимание, что мы здесь также явно создали стандартный менеджер **objects**. Дело в том, что когда в модели не определено никаких менеджеров записей, то автоматически создается **objects**. Как только мы определили свой собственный, то **objects** перестает существовать. Если он нужен, то следует также явно его прописать в модели.

Теперь в функции представления **index()** мы можем получать список опубликованных постов, используя новый менеджер **published**:

```
def index(request):
    posts = Women.published.all()
    data = {
        'title': 'Главная страница',
        'menu': menu,
        'posts': posts,
    }

    return render(request, 'women/index.html',
context=data)
```

А в шаблоне **index.html** условие проверки публикации можно убрать.

В функции представления **show_category()** пока будем читать все опубликованные статьи:

```
def show_category(request, cat_id):
    data = {
        'title': 'Отображение по рубрикам',
        'menu': menu,
        'posts': Women.published.all(),
        'cat_selected': cat_id,
    }

    return render(request, 'women/index.html',
context=data)
```

И удалим из файла **views.py** коллекцию **data_db**, так как теперь вся информация по постам берется непосредственно из БД.

Перечисляемое поле

Теперь добавим класс перечисления для поля **is_published**. Сейчас нам приходится прописывать 0, если статья не опубликована (черновик) и 1, если опубликована. Но оперировать числами не очень удобно, так как легко забыть, что значит 1 и 0 в данном случае. Гораздо лучше использовать осмысленные имена. Более того, в Django имеются классы, которые автоматизируют процесс создания подобных перечислений специально для

моделей. Подробную информацию о них можно посмотреть на странице документации:

<https://docs.djangoproject.com/en/4.2/ref/models/fields/#enumeration-types>

- **IntegerChoices** – для числовых наборов;
- **TextChoices** – для строковых наборов.

Давайте воспользуемся классом **IntegerChoices** для определения осмысленных имен опубликованных и черновых статей. Для этого в модели **Women** вначале объявим класс с именем **Status**, а затем применим его к полю **is_published** следующим образом:

```
class Women(models.Model):
    class Status(models.IntegerChoices):
        DRAFT = 0, 'Черновик'
        PUBLISHED = 1, 'Опубликовано'

    title = models.CharField(max_length=255)
    slug = models.SlugField(max_length=255,
db_index=True, unique=True)
    content = models.TextField(blank=True)
    time_create =
models.DateTimeField(auto_now_add=True)
    time_update = models.DateTimeField(auto_now=True)
    is_published =
models.BooleanField(choices=Status.choices,
default=Status.DRAFT)

    objects = models.Manager()
    published = PublishedModel()

    class Meta:
        ordering = ['-time_create']
        indexes = [
            models.Index(fields=['-time_create']),
        ]

    def get_absolute_url(self):
        return reverse('post', kwargs={'post_slug':
self.slug})

    def __str__(self):
        return self.title
```

Класс **Status** наследуется от класса **IntegerChoices**, который определяет своего рода перечисление на уровне фреймворка Django. В этом перечислении у нас два элемента с именами **DRAFT** и **PUBLISHED**. Эти атрибуты должны быть описаны как кортежи, состоящие из двух элементов: первое значение – это то, которое записывается в БД, а второе – это его имя (метка). В частности, имена используются в админ-панели Django.

После создания класс **Status** будет содержать атрибуты **DRAFT** и **PUBLISHED** со значениями 0 и 1, а также ряд вспомогательных атрибутов. Затем, с помощью параметра **choices** мы задаем виджет выбора значений для поля **is_published** с начальным значением (параметр **default**). Этот виджет будет использоваться при отображении данного поля на формах.

Так как модель **Women** поменялась, то нужно создать миграцию для изменения таблицы в БД:

```
python manage.py makemigrations
```

и применить их:

```
python manage.py migrate
```

Посмотрим, как это все будет работать. Перейдем в консоль фреймворка Django:

```
python manage.py shell_plus --print-sql
```

После этого прочитаем список всех статей:

```
w = Women.objects.all()
```

и установим поле **is_published** в значение **PUBLISHED**:

```
w.update(is_published=Women.Status.PUBLISHED)
```

Если отдельно проанализировать класс **Status**, то, во-первых, он имеет атрибут **choices** со списком кортежей:

```
Women.Status.choices
```

то увидим:

```
[(0, 'Черновик'), (1, 'Опубликовано')]
```

Атрибут **values** со списком значений:

```
Women.Status.values
```

получим:

[0, 1]

и атрибут **labels** со списком меток:

```
Women.Status.labels
```

возвратит список:

['Черновик', 'Опубликовано']

Благодаря введению класса-перечисления код стал несколько понятнее при использовании поля **is_published**.

Выйдем из консоли и в менеджере **PublishedModel** также воспользуемся введенным перечислением:

```
class PublishedModel(models.Manager):
    def get_queryset(self):
        return
super().get_queryset().filter(is_published=Women.Status
.PUBLISHED)
```

В результате получился более красивый программный код.

Задание

1. Создать таблицу БД, заполнить несколькими записями с помощью фреймворка.
2. Зайти в БД с помощью менеджера SQLiteStudio и убедиться, что данные в таблице действительно появились
3. Обеспечить наполнение страницы информацией из базы данных.
4. Реализовать работу с записями БД из своей программы: создание, изменение, удаление, выборку, фильтрацию, сортировку. Проверить, что все эти операции работают корректно.
5. Добавить в БД слаг и обеспечить отображение записей по их слагу.
6. Создать пользовательский менеджер модели.
7. Использовать в программе класс перечисления.