

A dark blue vertical bar is positioned on the left side of the page. A blue arrow-shaped banner points to the right, containing the year '2024 г.'. Below the banner, several thin, curved lines in shades of blue and grey sweep upwards from the bottom left corner.

2024 г.

Методическое пособие

*«Разработка веб-приложения с
использованием Python фреймворка Django»*

Алексеев И.И.

Содержание

<i>Глава 1. Серверная часть приложения</i>	<i>2</i>
<i>§1. Создание проекта Django</i>	<i>2</i>
<i>§2. Установка необходимых настроек.</i>	<i>2</i>
<i>§3. Создание первого приложения. Модель MTV.</i>	<i>5</i>
<i>§4. Основные HTML теги и структура HTML документа.....</i>	<i>11</i>
<i>§5. Написание класса модели. Знакомство с устройством баз данных SQL.</i>	<i>13</i>
<i>§6. Панель администрации. Получение данных из БД.</i>	<i>15</i>
<i>§7. Динамически-изменяемые страницы. Настройка админ-панели. Слагс.....</i>	<i>19</i>
<i>§8. Хранение медиа-файлов в БД. Внешние ключи.....</i>	<i>24</i>
<i>§9. Авторизация и регистрация пользователей.</i>	<i>27</i>
<i>§10. Оформление заказов. Работа с формами в Django</i>	<i>36</i>
<i>§11. Отправка электронных писем в Django</i>	<i>45</i>
<i>§12. Страница с заказами пользователей.....</i>	<i>47</i>
<i>§13. Перенос базы данных.....</i>	<i>50</i>
<i>Глава 2. Клиентская часть приложения.</i>	<i>55</i>

Глава 1. Серверная часть приложения

§1. Создание проекта Django

1.1 Создаем новый проект в PyCharm

1.2 Удаляем файл main.py (он нам не нужен)

1.3 Открываем терминал и прописываем команды:

```
pip install django
pip install django-bootstrap5
pip install django-debug-toolbar
pip install psycpg2
```

1.4 После установки библиотек выполняем команду:

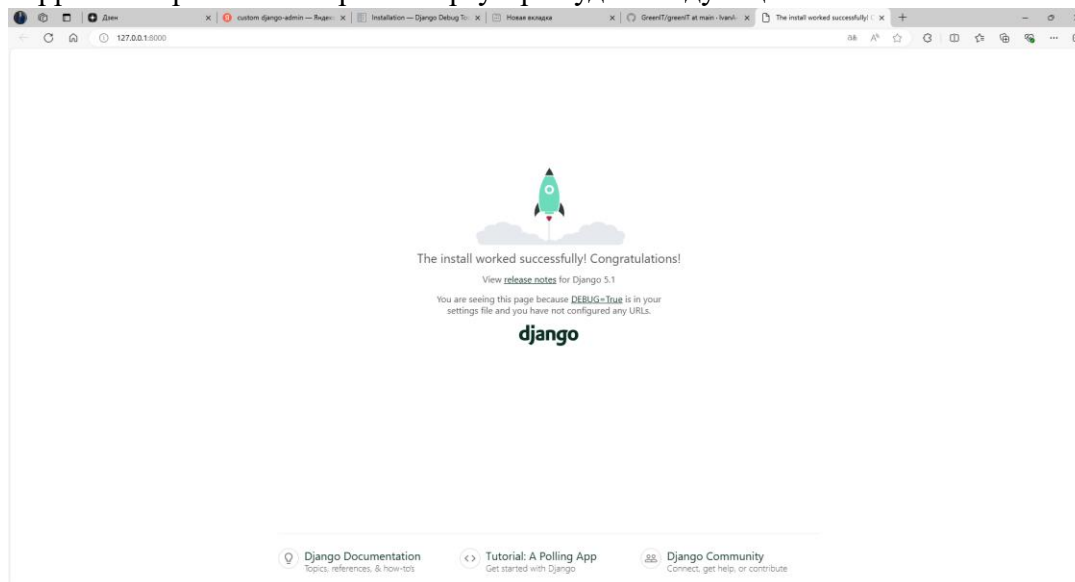
```
django-admin startproject <Имя проекта>
```

1.5 Далее выполняем команду `cd <Имя проекта>` и переходим в папку с проектом

1.6 Выполняем команды:

```
python manage.py migrate
python manage.py runserver
```

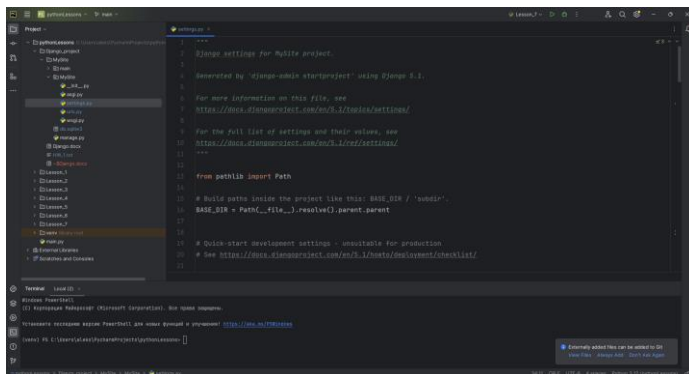
1.7 Переходим по появившейся в терминале ссылке и проверяем, что все работает. При корректной работе в открытом браузере будет следующее:



§2. Установка необходимых настроек.

После создания проекта на Django необходимо выполнить базовую настройку.

Переходим в папку проекта и открываем файл `settings.py`



В первую очередь установим русский язык и правильный часовой пояс. Для этого необходимо изменить значения переменных `LANGUAGE_CODE` на «ru» и `TIME_ZONE` на «Europe/Moscow».

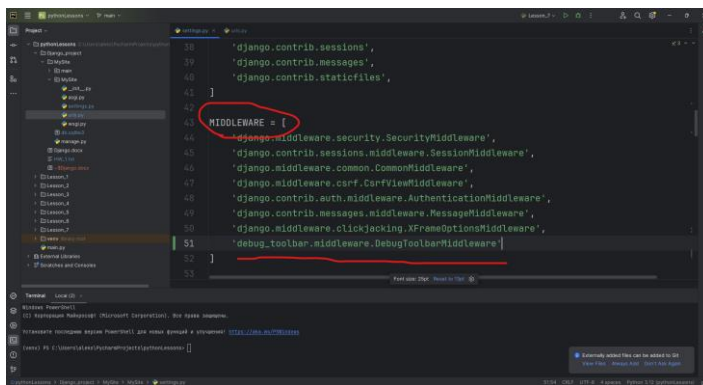
Далее установим зависимости для статических файлов и медиа(для хранения изображений, видео и т.п в БД).

Для этого после переменной `STATIC_URL = 'static/'` добавляем следующий код:

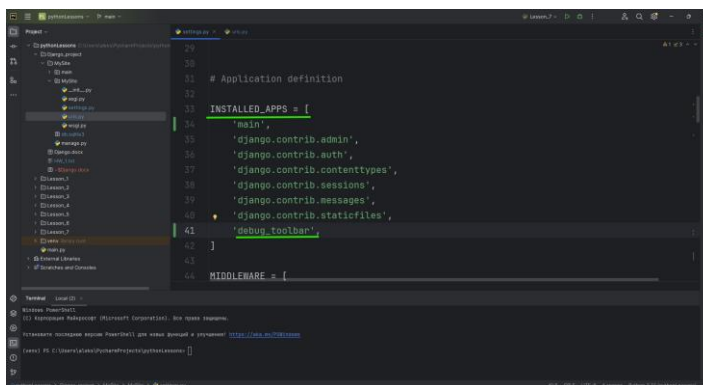
```
STATICFILES_DIRS = [  
    BASE_DIR / "static",  
]
```

```
MEDIA_ROOT = BASE_DIR / 'media'
```

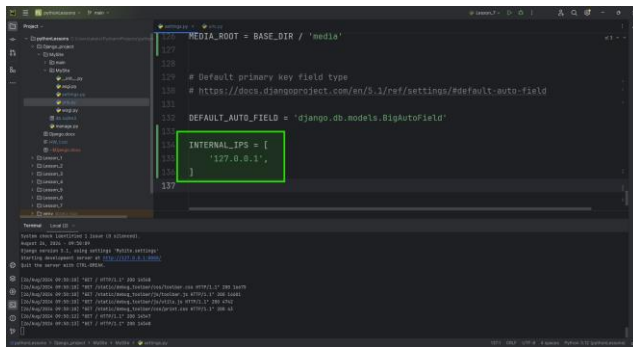
Теперь заранее позаботимся об установке отладочной панели Django, о которой будем вести разговор в дальнейшем. Для этого необходимо добавить `'debug_toolbar.middleware.DebugToolbarMiddleware'` в список `MIDDLEWARE`.



Далее добавляем «debug_toolbar» в список `INSTALLED_APPS`.



В конце файла добавляем следующий код



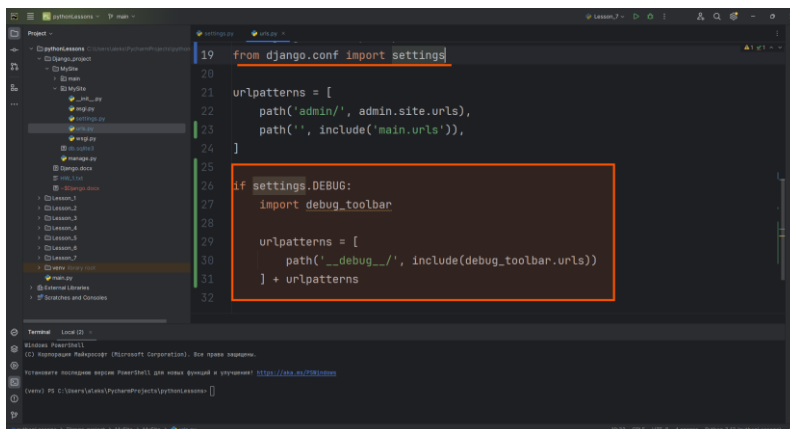
```
INTERNAL_IPS = [
    '127.0.0.1',
]
```

Теперь переходим в файл `urls.py` и добавляем следующий код в конец файла

```
from django.conf import settings

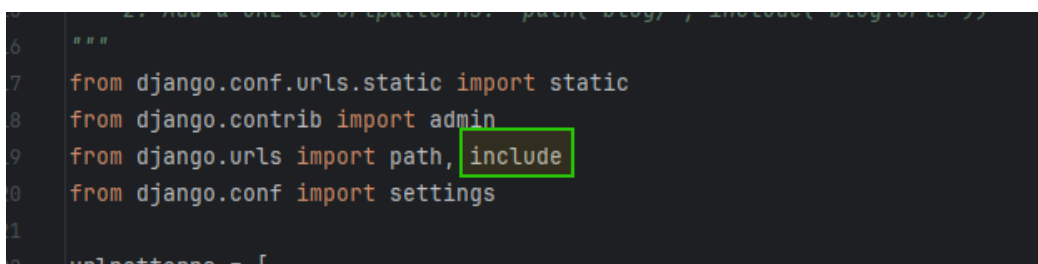
if settings.DEBUG:
    import debug_toolbar

urlpatterns = [
    path('__debug__/', include(debug_toolbar.urls))
] + urlpatterns
```



На этом первичные настройки завершены и можно переходить к дальнейшей работе над проектом.

Также необходимо импортировать `include` из `django.urls`:



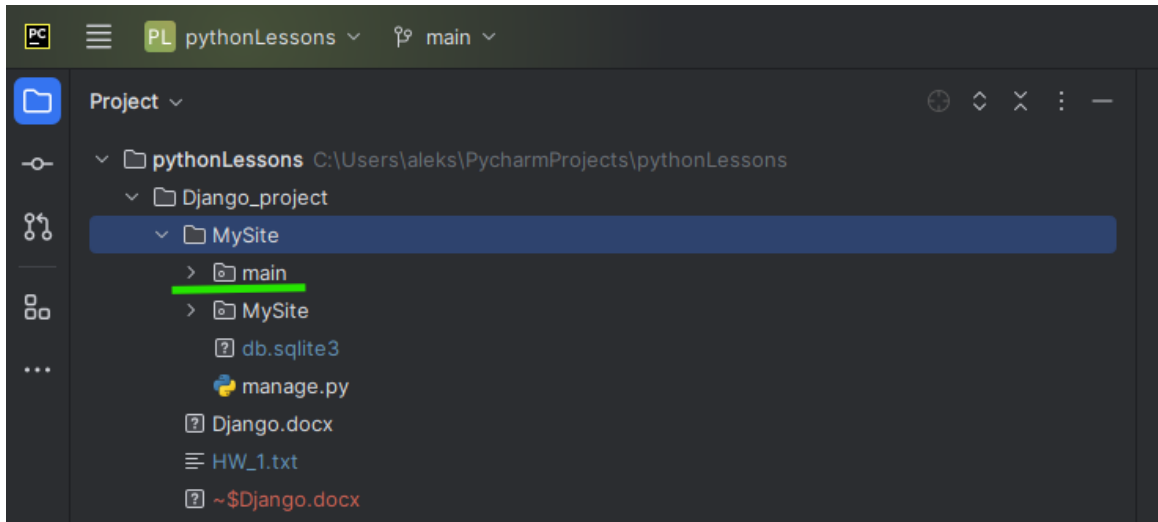
§3. Создание первого приложения. Модель MTV.

2.1 Для дальнейшей работы над проектом необходимо создать приложение

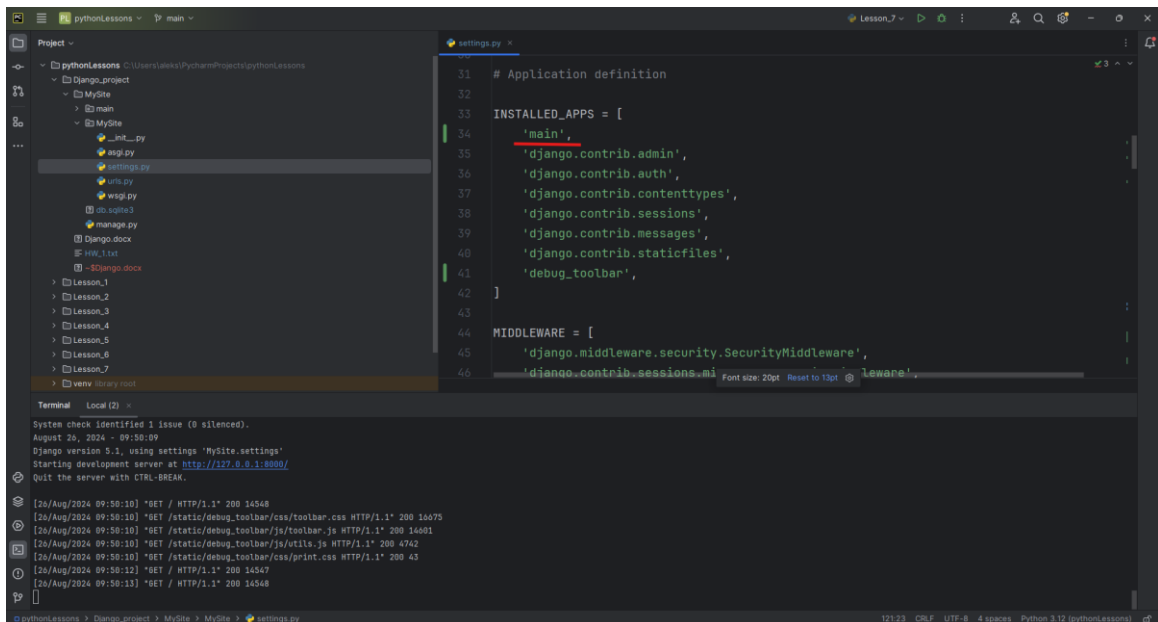
Для этого с использованием команд в терминале, описанных в блоке 1 переходим в папку проекта и выполняем следующую команду:

`python manage.py startapp <имя приложения>`

После выполнения команды в папке проекта будет создан каталог с именем приложения и основными файлами



Далее приложение необходимо зарегистрировать в проекте. Для этого переходим в файл `settings.py` и добавляем наше приложение в список `INSTALLED_APPS`



2.2 Модель MTV.

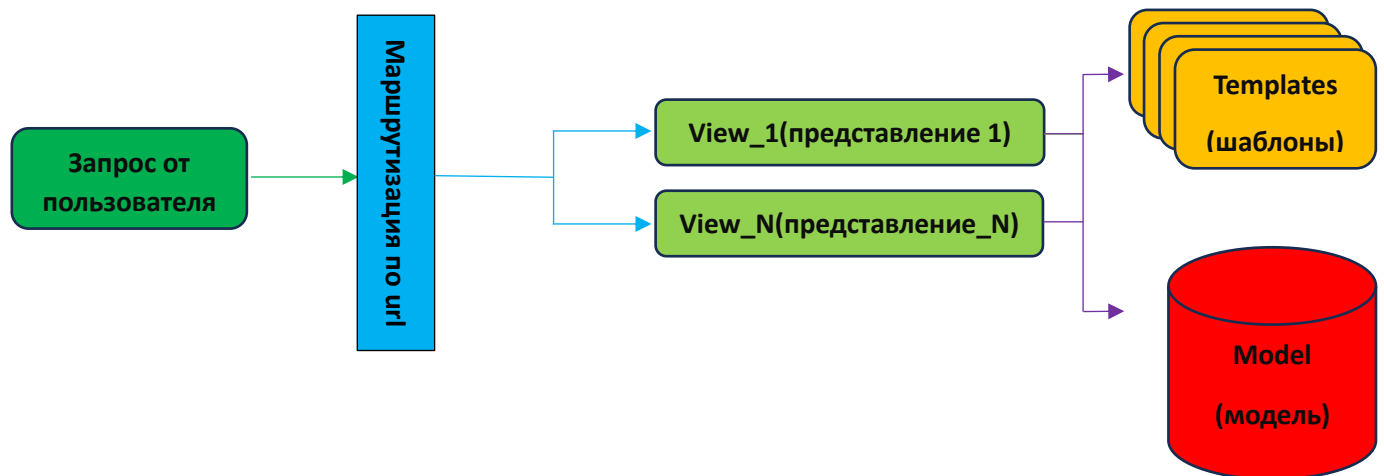
Django работает по модели MTV – Models Templates Views.

Models – модели

Templates – шаблоны

Views – представления

Далее изобразим схему примерной работы приложения и поговорим про каждый пункт отдельно



Работа приложения происходит по следующей схеме. От пользователя приходит запрос к приложению, который направляется с помощью маршрутизации по url-адресам, о которых мы поговорим далее. Далее обработкой данного запроса занимаются представления. Представления, сопоставляя url-адреса ищут в списке шаблонов необходимый, при необходимости обращаются к базе данных, извлекая оттуда данные, которые подразумевает запрос, после чего на основе шаблона и данных, взятых из БД представление формирует HTML документ, который и возвращает пользователю в виде страницы в браузере. Шаблон представляет из себя HTML документ, который отображает разметку определенной страницы в браузере, в которую встраиваются посредством специального алгоритма – шаблонизатора данные, которые были сгенерированы в приложении или взяты из БД. Представление – некоторая функция, которая занимается обработкой запроса от пользователя. Чуть сложнее дело обстоит с моделями. О них более подробно мы поговорим далее, пока достаточно знать, что модель является представлением таблицы в базе данных.

2.3 url-адреса и первый шаблон.

Теперь, когда мы ознакомились с тем, как работает наш проект – мы готовы написать наш первый шаблон и отобразить его в браузере. Для начала напишем функцию представления `index`, которая будет отвечать за отображение главной странички нашего сайта. Открываем файл `views.py` в папке нашего приложения, которое мы ранее создали и помещаем туда следующий код

```

1 from django.shortcuts import render
2
3
4 1 usage
5 def index(request):
6     return render(request, template_name='main/index.html')

```

```

def index(request):
    return render(request, 'main/index.html')

```

Готово. Функция представления написана и теперь нам необходимо создать соответствующий шаблон. Для этого мы создадим папку `templates` в папке нашего приложения, а в ней папку, название которой совпадает с названием нашего приложения. Это необходимо сделать для того, чтобы фреймворк мог разграничить, к какому приложению относятся те или иные шаблоны.

Прежде чем познакомиться с шаблонизатором и написать шаблон необходимо обсудить еще один немаловажный момент. Зачастую на страницах нашего сайта будут присутствовать такие элементы, которые должны быть видны, независимо от того, на какой странице сайта мы бы не находились. Для того, чтобы избежать написания одной и той же разметки каждый раз – создадим так называемый базовый шаблон `base.html`, который будет определять общую структуру и разметку всех страниц сайта, а далее будем наследовать данный шаблон и добавлять в него с помощью шаблонизатора информацию, которая должна отображаться на той или иной конкретной странице.

Создадим файл `base.html` в папке `templates/main` и разместим в нем следующий код

```

views.py  <> base.html x
1 <!doctype html>
2 <html lang="ru">
3     <head>
4         <meta charset="UTF-8">
5         <meta name="viewport"
6             content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
7         <meta http-equiv="X-UA-Compatible" content="ie=edge">
8         <title>
9             {% block title %}{% endblock %}
10        </title>
11    </head>
12
13    <body>
14        <a href="{% url 'main:index' %}">Главная</a>|
15
16        {% block content %}{% endblock %}
17    </body>
18 </html>

```

```

<!doctype html>
<html lang="ru">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport"
            content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-
scale=1.0, minimum-scale=1.0">
        <meta http-equiv="X-UA-Compatible" content="ie=edge">

```



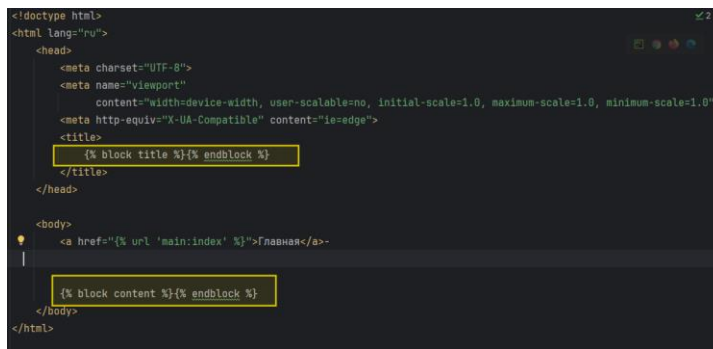
```

<title>
    {% block title %} {% endblock %}
</title>
</head>

<body>
    <a href="{% url 'main:index' %}">Главная</a>-
    {% block content %} {% endblock %}
</body>
</html>

```

Пока не будем подробно останавливаться на структуре HTML документа, поговорим об этом далее. Сейчас нам необходимо поговорить о том, что уже в данном шаблоне мы начинаем работу с шаблонизатором, а именно вот здесь



Вместо того, чтобы указать заголовок страницы или написать разметку для основного контента мы разместим в тегах title и body так называемые блоки. Это будет работать следующим образом: та разметка, которую мы напишем для базового шаблона будет отображаться на всех страницах, которые унаследуют базовый шаблон, а для каждой отдельной страницы информация, которая относится только к ней будет вынесена в отдельный блок для каждого шаблона и в дальнейшем встроится в базовый шаблон в тех местах, где мы создали блоки. Будьте внимательны – блок необходимо закрывать после объявления: `{% block content %} {% endblock %}`. Открытие блока начинается с ключевого слова `block`, а далее следует название блока. В шаблонизаторе большинство конструкций будет заключено в операторные скобки `{% %}`. Также будут встречаться и другие операторные скобки, но о них мы поговорим позже.

Теперь, когда базовый шаблон определен мы можем перейти к написанию шаблона главной страницы сайта. Для этого создадим все в той же папке файл `index.html`.

В первую очередь нам необходимо унаследовать базовый шаблон. Делается это с помощью следующего кода

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

Отлично. Теперь можно перейти к написанию дальнейшей разметки главной страницы. Пока мы разместим на ней всего лишь один заголовок первого уровня с надписью **Главная** и добавим заголовок страницы.

Поскольку мы унаследовали базовый шаблон – нам доступны блоки, которые мы создали в нем. Разместим в них необходимую информацию.

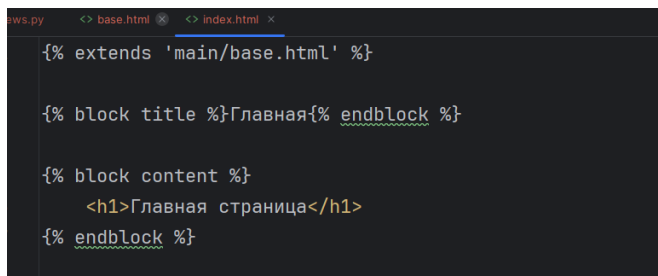
```
{% block title %}Главная{% endblock %}
```

```
{% block content %}
```

```
    <h1>Главная страница</h1>
```

```
{% endblock %}
```

Таким образом код главной страницы на данный момент будет выглядеть следующим образом



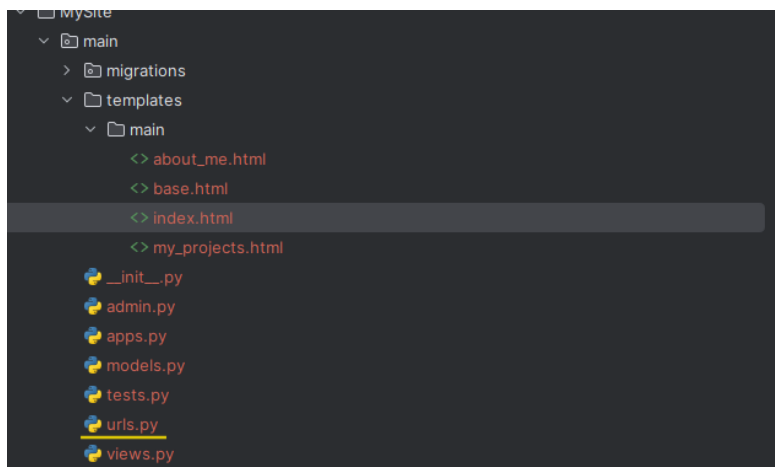
```
{% extends 'main/base.html' %}

{% block title %}Главная{% endblock %}

{% block content %}
    <h1>Главная страница</h1>
{% endblock %}
```

```
{% extends 'main/base.html' %}
{% block title %}Главная{% endblock %}
{% block content %}
    <h1>Главная страница</h1>
{% endblock %}
```

Осталось совсем немного. Необходимо задать url адрес, по которому будет доступна главная страница. Для этого создадим в папке приложения файл urls.py.



В этом файле в первую очередь нам необходимо импортировать функцию path из django.urls и файл, содержащий функции представления

```
from django.urls import path
```

```
from . import views
```

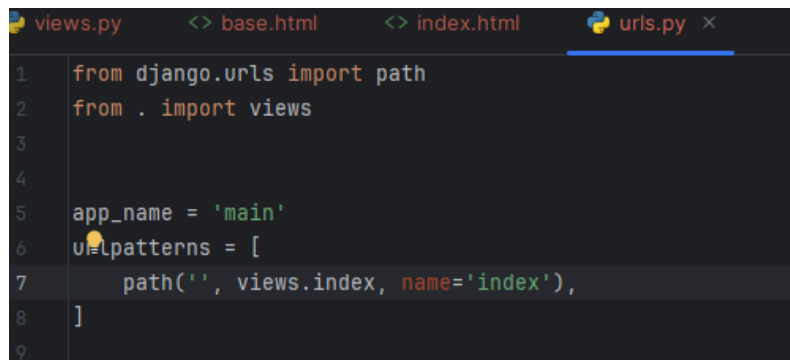
Теперь для дальнейшего удобства при обращении к url адресам данного приложения – укажем его название в переменной `app_name`

```
app_name = 'main'
```

Осталось лишь задать список, содержащий url-адреса данного приложения и почти все будет готово

```
urlpatterns = [  
    path('', views.index, name='index'),  
]
```

Размещаем в списке первый url-адрес с помощью функции `path`, в которую передаем три аргумента – сам адрес, функцию представления, имя, по которому будет происходить обращение к данному url. Таким образом имеем следующий код

A screenshot of a code editor with four tabs: 'views.py', 'base.html', 'index.html', and 'urls.py'. The 'urls.py' tab is active and shows the following Python code:

```
1 from django.urls import path  
2 from . import views  
3  
4  
5 app_name = 'main'  
6 urlpatterns = [  
7     path('', views.index, name='index'),  
8 ]  
9
```

```
from django.urls import path  
from . import views
```

```
app_name = 'main'  
urlpatterns = [  
    path("", views.index, name='index'),  
]
```

Ну вот почти все и готово. Осталось лишь добавить url-адреса из нашего приложения в основной файл, чтобы они были видны в проекте. Для этого открываем файл `urls.py` в папке нашего проекта и добавляем в список `urlpatterns` следующий элемент

```
path('', include("main.urls"))
```

Добавляем `include` в `import`, который присутствует в данном файле. Имеем следующее

```

from django.contrib import admin
from django.urls import path, include
from django.conf import settings

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('main.urls')),
]

if settings.DEBUG:
    import debug_toolbar

    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls))
    ] + urlpatterns

```

Отлично. Теперь можно перейти в терминал и выполнить уже знакомую команду `python manage.py runserver`, чтобы посмотреть, как работает наше приложение. Если все было написано правильно – в браузере должна отобразиться страница с названием Главная, наверху которой размещена ссылка Главная, а далее заголовок первого уровня Главная страница.

§4. Основные HTML теги и структура HTML документа

Поскольку в дальнейшем нам предстоит написание шаблонов – не будет лишним ознакомиться с основными HTML тегами и структурой HTML документа.

Начнем со второго. HTML документ представляет из себя разметку страницы в браузере и состоит из нескольких блоков. Любой HTML документ начинается с обязательных тегов

```
<!doctype html>
```

```
<html>
```

```
</html>
```

Теги в HTML бывают парные и непарные. Отличие состоит в том, что парные теги требуют закрытия, а непарные – нет:

`<tag>` - непарный тег

`<tag></tag>` - парный тег

Основными блоками в HTML документе являются блоки, заключенные в тегах `<head></head>` и `<body></body>`

В теге `<head>`, как правило размещают настройки страницы и информацию, характеризующую страницу, например название страницы, а также подключение дополнительных источников данных. Например:

```
<!doctype html>
```

```
<html lang="ru">
```

```
<head>
  <meta charset="UTF-8">
  <meta name="viewport"
    content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-
scale=1.0, minimum-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>My site</title>
</head>

<body></body>
</html>
```

В теге <body> размещается основное содержимое страницы.

Теперь пробежимся по списку основных тегов.

<title></title> - заголовок страницы

<p></p> - абзац с текстом

<h1></h1>...<h6></h6> - заголовки уровня H1 – H6. Чем выше уровень тем меньше шрифт.

 - гиперссылка. В атрибуте href необходимо указать url-адрес, по которому будет осуществлен переход при нажатии на гиперссылку.

<button></button> - кнопка

 - отступ на новую строку

 - нумерованный список

 - элемент списка

 - маркированный список

 - текстовый контейнер

<div></div> - контейнер. Чаще всего используется для группировки элементов.

<link> - позволяет подключать внешние источники информации

 - добавляет изображение. В атрибуте src указывается источник.

<script></script> - позволяет подключить скрипты

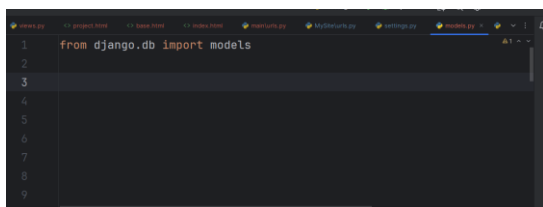
Пока что нам будет достаточно данного набора тегов для дальнейшей работы. В случае необходимости в дальнейшем мы изучим и другие теги.

§5. Написание класса модели. Знакомство с устройством баз данных SQL.

SQL – structure query language – язык запросов, позволяющий производить манипуляции с реляционными базами данных. По своей структуре база данных представляет из себя набор таблиц, в которых хранятся данные. Каждая ячейка такой таблицы называется полем. Посредством запросов можно добавлять, удалять, изменять данные и таблицы, а также извлекать данные из таблиц. В Django нам не придется самим писать SQL-запросы, за нас это сделает фреймворк. При работе над проектом базу данных и ее структуру желательно планировать заранее, поскольку изменить таблицы в дальнейшем может быть проблематично. Таблицы в базе данных представляются в Django с помощью моделей. Модель – некоторый класс, описывающий таблицу и ее поля в БД. Э

Давайте напишем первую модель Projects, в которой будут храниться данные о нашем портфолио – работах, которые мы уже выполнили и хотим выложить на сайт для того, чтобы показать пользователю.

Откроем файл models.py в папке нашего приложения.



В данном файле нам необходимо создать класс модели Projects, который будет отвечать за хранение информации о наших работах. В учебных целях мы в дальнейшем будем изменять модели. На практике так делать не рекомендуется.

Добавим в нашу модель следующие поля:

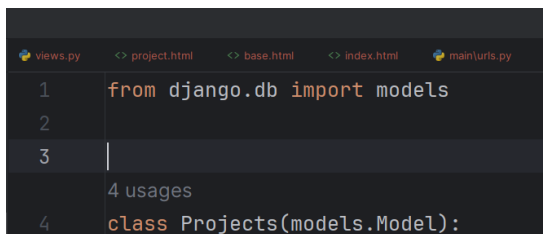
project_name(CharField) – название работы

description(TEXTFIELD) – описание

date_added(DATETIMEFIELD) – дата и время публикации

price(FLOATFIELD) – цена

Поля таблиц в БД также имеют свой тип данных, который необходимо будет указать. Также необходимо отметить, что при создании модели стоит указать, что любой класс модели наследует класс models.Model



Теперь укажем наши поля

```

from django.db import models

4 usages
class Projects(models.Model):
    project_name = models.CharField(max_length=250, verbose_name='Название проекта',
                                    blank=True, null=True)
    description = models.TextField(verbose_name='Описание', blank=True, null=True)
    date_added = models.DateTimeField(auto_now=True, verbose_name='Дата добавления')
    price = models.FloatField(verbose_name='Цена', blank=True, null=True)

```

Для полей также укажем некоторые атрибуты. У поля `project_name`, которое является CHARFIELD необходимо указать обязательный атрибут `max_length`, определяющий максимально возможную длину названия проекта. Также укажем атрибуты `verbose_name`, `blank=True`, `null=True`.

Об атрибуте `verbose_name` поговорим далее. Атрибуты `blank` и `null` в значении `True` позволят нам изменять модель, когда в ней уже будут находиться внесенные в БД данные. Эти же атрибуты будем указывать и для всех остальных полей модели. Делаем это в учебных целях, поскольку будем изменять модель с уже имеющимися данными. На практике так делать не стоит и БД необходимо планировать полностью изначально.

Также сразу добавим в модель так называемый магический метод «`__str__`»:

```

def __str__(self):
    return self.project_name

```

```

def __str__(self):
    return self.project_name

```

О назначении этого метода уже очень скоро мы поговорим, как и о так называемом метаклассе, который сейчас и напишем. Добавляем в класс модели следующий код:

```

class Meta:
    verbose_name = 'Проект'
    verbose_name_plural = 'Проекты'

```

```

class Meta:
    verbose_name = 'Проект'
    verbose_name_plural = 'Проекты'

```

Отлично. Наша модель написана и теперь ее код выглядит следующим образом

```

from django.db import models

4 usages
class Projects(models.Model):
    project_name = models.CharField(max_length=250, verbose_name='Название проекта',
                                    blank=True, null=True)
    description = models.TextField(verbose_name='Описание', blank=True, null=True)
    date_added = models.DateTimeField(auto_now=True, verbose_name='Дата добавления')
    price = models.FloatField(verbose_name='Цена', blank=True, null=True)

    def __str__(self):
        return self.project_name

    class Meta:
        verbose_name = 'Проект'
        verbose_name_plural = 'Проекты'

```

```

class Projects(models.Model):
    project_name = models.CharField(max_length=250, verbose_name='Название проекта',
                                    blank=True, null=True)
    description = models.TextField(verbose_name='Описание', blank=True, null=True)
    date_added = models.DateTimeField(auto_now=True, verbose_name='Дата добавления')
    price = models.FloatField(verbose_name='Цена', blank=True, null=True)

    def __str__(self):
        return self.project_name

class Meta:
    verbose_name = 'Проект'
    verbose_name_plural = 'Проекты'

```

Теперь нам необходимо перенести нашу модель в базу данных и создать на основе нее таблицу. Заходим в терминал и пишем команды

```
python manage.py makemigrations
```

```
python manage.py migrate
```

Они запускают миграции и синхронизируют изменения в проекте и базу данных.

§6. Панель администрации. Получение данных из БД.

У Django есть своя административная панель, которая позволяет удобно управлять сайтом. Давайте познакомимся с ней. Для начала создадим администратора (суперпользователя) с помощью команды

```
python manage.py createsuperuser
```

После этого запустим сервер и откроем сайт. В адресной строке допишем /admin



Теперь мы попадаем в панель администрации, где необходимо ввести логин и пароль, которые мы указывали при создании суперпользователя

 A screenshot of the Django Admin login page. At the top, there is a blue header bar with the text "Администрирование Django" and a small circular icon. Below the header, there are two input fields: the first is labeled "Имя пользователя:" and the second is labeled "Пароль:". Below these fields is a blue button with the text "Войти".

После авторизации мы попадаем в админ-панель.

Теперь сделаем так, чтобы наша модель отображалась в админ-панели. Для этого откроем файл `admin.py` и зарегистрируем нашу модель в панели администрации. Это делается добавлением следующего кода:

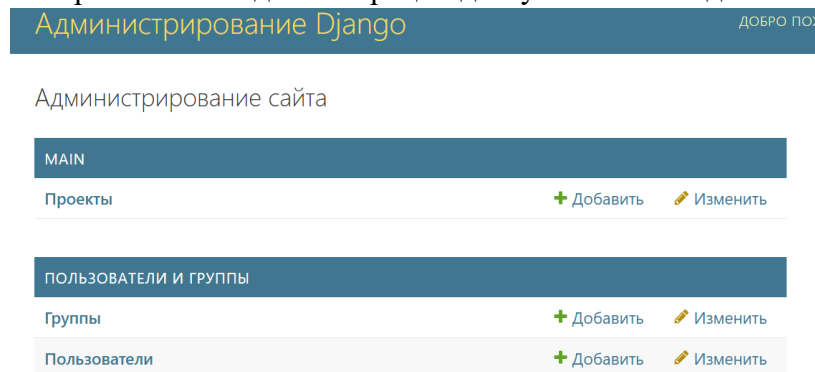
```
admin.site.register(<Имя модели>)
```

Также необходимо импортировать файл, в котором хранятся модели.

```
from django.contrib import admin
from . import models

admin.site.register(models.Projects)
```

Теперь в панели администрации доступна наша модель



При добавлении данных через панель администрации – они автоматически будут добавляться и в базу данных. Теперь поговорим о метаклассе, который мы добавили в класс модели. Метакласс определяет поведение модели в панели администрации. `verbose_name` и `verbose_name_plural` отвечают за правильное название модели во множественном и единственном числе в панели администрации. Атрибут `verbose_name` в полях модели выполняет ту же функцию.

Добавить Проект

Название проекта:

Описание:

Цена:

Д/ДТ

Теперь напишем шаблон, который будет отображать страницу с проектами. Для этого перейдем в файл `views.py` и поместим туда следующий код

```
def my_projects(request):
    projects = Projects.objects.order_by('-date_added')

    context = {'projects': projects}

    return render(request, 'main/my_projects.html', context)
```

```
def my_projects(request):
    projects = Projects.objects.order_by('-date_added')

    context = {'projects': projects}

    return render(request, 'main/my_projects.html', context)
```

Давайте познакомимся с ним более подробно. В отличие от других функций представления, здесь нам необходимо получать данные из БД, которые будут размещаться на соответствующей странице сайта. Для получения всех проектов из БД необходимо обратиться к модели `Projects`. Полученные данные будем сортировать таким образом, чтобы самые свежие по дате и времени добавления оказались выше. В этом нам поможет фильтр `order_by`.

```
projects = Projects.objects.order_by('-date_added')
```

Теперь необходимо передать полученные данные в наш шаблон. Для этого создадим словарь и передадим его вместе с ответом на запрос.

```
context = {'projects': projects}
return render(request, 'main/my_projects.html', context)
```

Теперь перейдем в файл `urls.py` и сопоставим url-адреса.

```
path('my_projects', views.my_projects, name='projects'),
```

Не забудьте при написании функции представления импортировать модели:

```
from .models import *
```

Теперь создадим в папке с шаблонами файл `my_projects.html` и поместим в него следующий код

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

{% block title %}Мои работы{% endblock %}

{% block content %}
    <h1>Мои работы</h1>
    <ul>
        {% for project in projects %}
            <li>
                <h4>Date added: {{project.date_added}}</h4>
                <a href="">{{project}}</a>

            </li>
            {% empty %}
            <li>
                <h3>На данный момент нет проектов</h3>
            </li>
        {% endfor %}

    </ul>
{% endblock %}
```

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

{% block title %}Мои работы{% endblock %}

{% block content %}
    <h1>Мои работы</h1>
    <ul>
        {% for project in projects %}
            <li>
                <h4>Date added: {{project.date_added}}</h4>
                <a href="">{{project}}</a>

            </li>
            {% empty %}
            <li>
                <h3>На данный момент нет проектов</h3>
            </li>
        {% endfor %}

    </ul>
{% endblock %}
```

Здесь мы знакомимся с новыми возможностями шаблонизатора – циклами и обращением к данным, полученными из БД.

В цикле, расположенном внутри маркированного списка, мы перебираем все элементы в списке данных, полученных из БД, и выводим их на страницу.

```
<ul>
    {% for project in projects %}
        <li>
            <h4>Date added: {{project.date_added}}</h4>
```

```

<a href="">{{project}}</a>

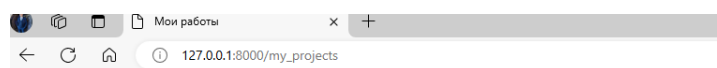
</li>
{% empty %}
<li>
    <h3>На данный момент нет проектов</h3>
</li>
{% endfor %}

```

```
</ul>
```

Информация в блоке `{% empty %}` будет отображаться в случае, если данных из БД не было получено и список пуст. Обращение к элементам, полученным из БД, происходит по полям, которые мы задавали в классе модели. Для получения названия проекта достаточно обратиться к самому элементу без указания поля. В этом нам помогает метод `__str__`, который возвращает название проекта. Будьте внимательны, вывод некоторой информации в шаблонизаторе производится в следующих операторных скобках `{{ }}`

Теперь можем открыть наш сайт и посмотреть, что получилось.



[Главная](#)- [Обо мне](#)- [Мои работы](#)

Мои работы

- Date added: 28 августа 2024 г. 19:41

[Глиняный горшок](#)

- Date added: 28 августа 2024 г. 19:34

[Светильник из дерева](#)

§7. Динамически-изменяемые страницы. Настройка админ-панели. Слаг.

В прошлом блоке мы написали шаблон, отображающий страницу с проектами. В том шаблоне мы сделали каждый проект на странице в виде ссылки. Теперь сделаем так, чтобы каждый проект можно было открыть на отдельной странице. Поскольку проектов в БД может быть неограниченное количество – было бы нелогично писать для каждого отдельный шаблон. Ввиду этого напишем один шаблон, который будет изменяться в зависимости от выбранного проекта. Для этого перейдем в файл `views.py` и напишем следующую функцию представления:

```

usage
def project_view(request, project_id):
    project = Projects.objects.get(id=project_id)

    context = {'project': project}

    return render(request, template_name='main/project.html', context)

```

```

def project_view(request, project_id):
    project = Projects.objects.get(id=project_id)

    context = {'project': project}

    return render(request, 'main/project.html', context)

```

Она очень похожа на функцию, отображающую страницу проектов, но в отличие от нее принимает еще один параметр `project_id`. В базе данных в таблице по умолчанию присутствует поле `id`. По нему мы и будем получать конкретный проект. Для этого будем обращаться к модели следующим образом `project = Projects.objects.get(id=project_id)`. Далее мы передаем таким же образом полученный проект в шаблон. Теперь сопоставим url.

```

path('my_projects/<int:project_id>', views.project_view, name='project'),

```

Создаем шаблон `project.html` и пишем в нем следующий код.

```

{% extends 'main/base.html' %}

{% block title %}{{project}}{% endblock %}

{% block content %}
    <h1>{{project}}</h1>
    <h4>Date added: {{project.date_added}}</h4>
    <p>
        <strong>Description</strong>
        <br>
        {{project.description}}
    </p>
    <p>Price: {{project.price}}$</p>
{% endblock %}

```

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

{% block title %}{{project}}{% endblock %}

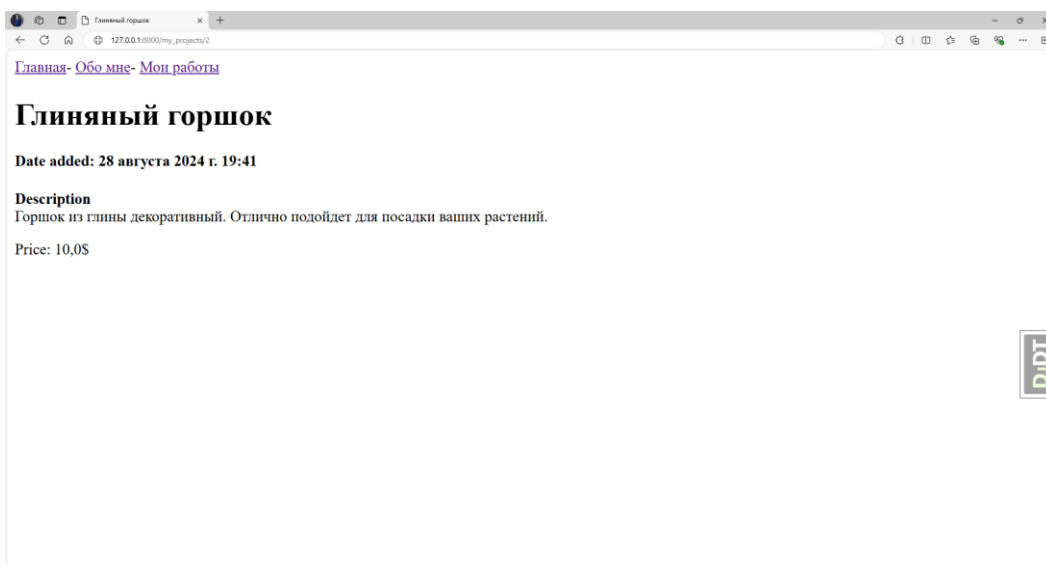
{% block content %}
    <h1>{{project}}</h1>
    <h4>Date added: {{project.date_added}}</h4>
    <p>
        <strong>Description</strong>
        <br>
        {{project.description}}
    </p>
    <p>Price: {{project.price}}$</p>
{% endblock %}
```

Теперь осталось лишь добавить url-адрес в ссылку в файле my_projects.html

```
<> base.html <> index.html main\urls.py MySite\urls.py settings.py models.py

1 {% extends 'main/base.html' %}
2
3 {% block title %}Мои работы{% endblock %}
4
5 {% block content %}
6     <h1>Мои работы</h1>
7     <ul>
8         {% for project in projects %}
9         <li>
10             <h4>Date added: {{project.date_added}}</h4>
11             <a href="{% url 'main:project' project.id %}">{{project}}</a>
12         </li>
13         {% empty %}
14         <li>
15             <h3>На данный момент нет проектов</h3>
16         </li>
17         {% endfor %}
18     </ul>
19
20 {% endblock %}
21
```

Тут мы как раз и передаем в функцию представления id проекта. Отлично. Теперь наши ссылки стали активны.



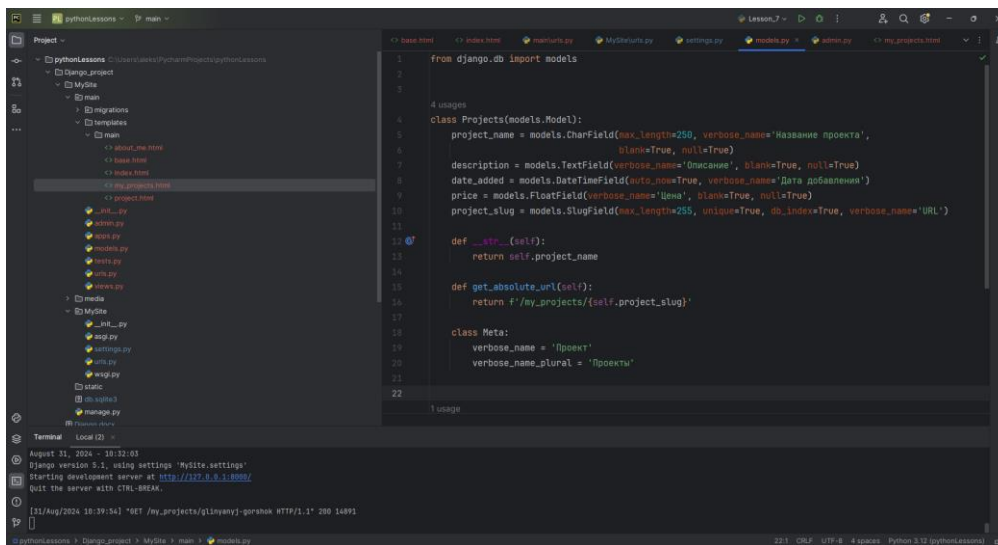
Обращаться в url по id не очень удобно и безопасно. Поэтому сделаем так, чтобы у каждой динамически-изменяемой страницы в адресной строке было свое короткое имя, которое называется слаг. Для добавления слагов необходимо добавить в модель соответствующее поле.

```
project_slug = models.SlugField(max_length=255, unique=True, db_index=True, verbose_name='URL')
```

Также добавим еще один метод `get_absolute_url`, который в дальнейшем нам пригодится.

```
def get_absolute_url(self):  
    return f'/my_projects/{self.project_slug}'
```

Теперь модель выглядит так.

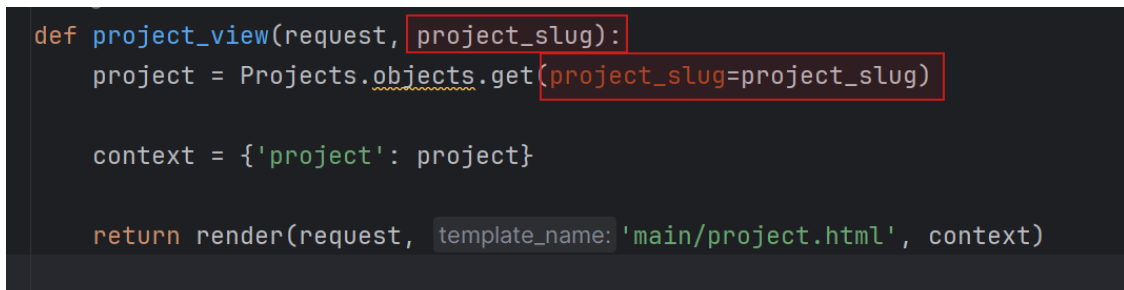


Выполняем миграции уже знакомыми командами

```
python manage.py makemigrations
```

```
python manage.py migrate
```

Теперь в файлах `views`, `urls`, `my_projects.html` следующие изменения, чтобы мы могли обращаться к динамически-изменяемым страницам по их слагу.



```

from django.urls import path
from . import views

app_name = 'main'
urlpatterns = [
    path('', views.index, name='index'),
    path('about_me', views.about_me, name='about_me'),
    path('my_projects', views.my_projects, name='projects'),
    path('my_projects/<slug:project_slug>', views.project_view, name='project'),
]

```

```

1 {% extends 'main/base.html' %}
2
3 {% block title %}Мои работы{% endblock %}
4
5 {% block content %}
6     <h1>Мои работы</h1>
7     <ul>
8         {% for project in projects %}
9             <li>
10                 <h4>Date added: {{project.date_added}}</h4>
11                 <a href="{% url 'main:project' project.project_slug %}">{{project}}</a>
12             </li>
13         {% empty %}
14             <li>
15                 <h3>На данный момент нет проектов</h3>
16             </li>
17         {% endfor %}
18     </ul>
19
20
21 {% endblock %}

```

Готово. Теперь у нас есть слаг. Далее настроим отображение модели в панели администрации. Для этого нам необходимо открыть файл `admin.py` и написать класс `ProjectsAdmin`, который определит отображение модели в панели администрации.

```

from . import models

1 usage
class ProjectsAdmin(admin.ModelAdmin):
    list_display = ['id', 'project_name', 'date_added']
    list_display_links = ['project_name']
    search_fields = ['project_name']
    prepopulated_fields = {"project_slug": ('project_name', )}
    ordering = ['id', 'date_added']

admin.site.register(models.Projects, ProjectsAdmin)

```

```

class ProjectsAdmin(admin.ModelAdmin):
    list_display = ['id', 'project_name', 'date_added']

```



```
list_display_links = ['project_name']
search_fields = ['project_name']
prepopulated_fields = {"project_slug": ('project_name', )}
ordering = ['id', 'date_added']
```

Список `list_display` отвечает за то, какие поля модели будут отображены в админ-панели.

Список `list_display_links` отвечает за то, какие поля модели в админ-панели будут активны

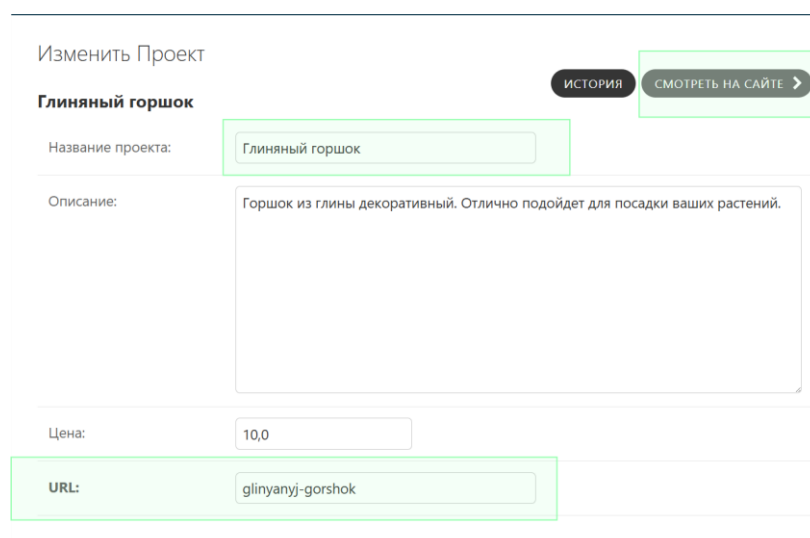
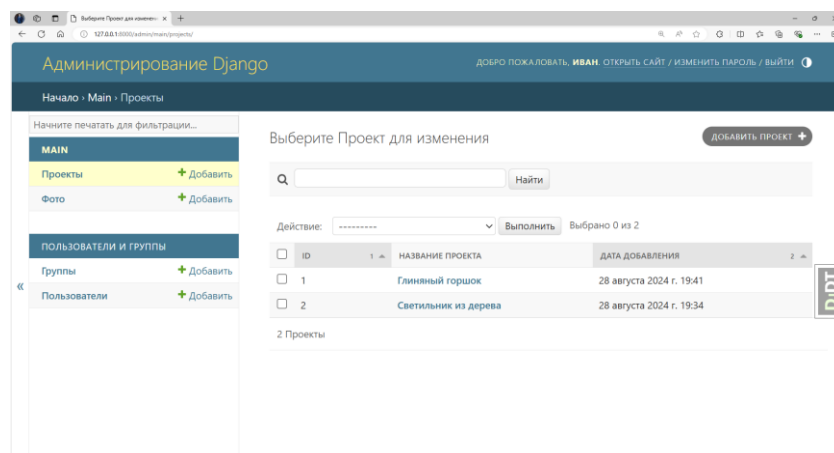
Список `search_fields` определяет поля, по которым можно вести поиск

Словарь `prepopulated_fields` отвечает за автоматическое формирование слага по названию проекта.

Список `ordering` отвечает за порядок отображения данных.

Для того, чтобы все работало – класс регистрируем вместе с моделью.

Также сейчас скажем о том, зачем был нужен метод `get_absolute_url`. Он позволит нам прямо из панели администрации перейти на страницу добавляемого проекта.



§8. Хранение медиа-файлов в БД. Внешние ключи.

На странице проекта удобно было бы располагать фото. Фото может быть разное количество и удобно было бы хранить их в базе данных. Это происходит таким

образом, что фото и другие медиа-материалы хранятся в отдельной папке, а в БД мы будем сохранять зависимости, которые будут указывать на нужный файл. Для хранения фото создадим отдельную модель. Для того, чтобы понимать, к какому проекту относятся те или иные фото – необходимо установить зависимости между таблицами. Для этого будем использовать так называемый внешний ключ FOREIGN_KEY, который будет указывать на id проекта, к которому относятся фото. Таким образом мы можем хранить неограниченное количество фото, привязанных к одному и тому же проекту. Код модели будет выглядеть следующим образом.

```
class ProjectPhoto(models.Model):
    project = models.ForeignKey(Projects, on_delete=models.CASCADE,
verbose_name='Проект')
    photo = models.ImageField(verbose_name='Фото', null=True,
upload_to='media/project_photo')

    class Meta:
        verbose_name = 'Фото'
        verbose_name_plural = 'Фото'
```

```
1 usage
class ProjectPhoto(models.Model):
    project = models.ForeignKey(Projects, on_delete=models.CASCADE, verbose_name='Проект')
    photo = models.ImageField(verbose_name='Фото', null=True, upload_to='media/project_photo')

    class Meta:
        verbose_name = 'Фото'
        verbose_name_plural = 'Фото'
```

Не забываем зарегистрировать модель в панели администрации, добавив `admin.site.register(models.ProjectPhoto)` в файл `admin.py`.

Для хранения фото используется тип поля `IMAGEFIELD`. Чтобы все корректно работало выполняем команду

```
pip install pillow
```

Далее выполняем миграции.

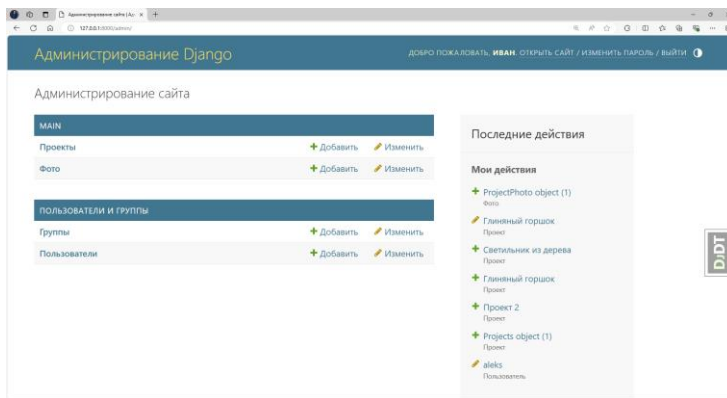
Атрибут `upload_to` определяет, куда будут сохраняться добавляемые файлы.

Внешний ключ задается следующим образом

```
project = models.ForeignKey(Projects, on_delete=models.CASCADE,
verbose_name='Проект')
```

Ссылается он на модель `Projects`, атрибут `on_delete=models.CASCADE` определяет каскадное удаление. Это означает, что при удалении записи из таблицы `Projects`, будут удалены все фото, связанные с этой записью внешним ключом.

Теперь можно зайти в панель администрации и посмотреть, что у нас получилось.



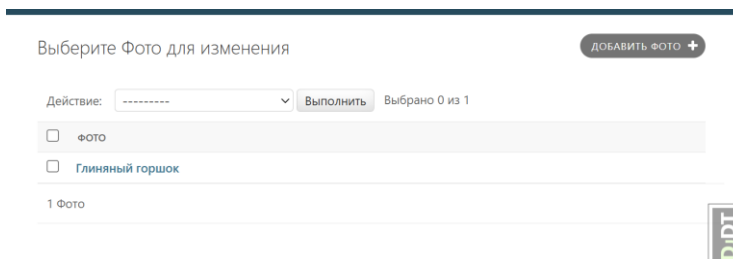
Добавим также метод `__str__`, который будет возвращать имя проекта, к которому относятся добавленные фотографии.

```
1 usage
class ProjectPhoto(models.Model):
    project = models.ForeignKey(Projects, on_delete=models.CASCADE, verbose_name='Проект')
    photo = models.ImageField(verbose_name='Фото', null=True, upload_to='media/project_photo')

    def __str__(self):
        return self.project.project_name

    class Meta:
        verbose_name = 'Фото'
        verbose_name_plural = 'Фото'
```

Готово. Все работает корректно.



Теперь напишем класс для корректного отображения модели в панели администрации.

```
1 usage
class ProjectPhotoAdmin(admin.ModelAdmin):
    list_display = ['project', 'get_html_photo']
    list_display_links = ['project']
    search_fields = ['project']
    ordering = ['project']
    readonly_fields = ('get_html_photo',)

    def get_html_photo(self, object):
        return mark_safe(f'')

    get_html_photo.short_description = 'Фото'
```

```
class ProjectPhotoAdmin(admin.ModelAdmin):
    list_display = ['project', 'get_html_photo']
    list_display_links = ['project']
    search_fields = ['project']
    ordering = ['project']
    readonly_fields = ('get_html_photo',)
```

```
def get_html_photo(self, object):
    return mark_safe(f'')
```

```
get_html_photo.short_description = 'Фото'
```

Не забываем добавить некоторые импорты и зарегистрировать класс.

```
from django.utils.safestring import mark_safe
```

```
admin.site.register(models.ProjectPhoto, ProjectPhotoAdmin)
```

Метод `get_html-photo` позволит нам просмотреть превью добавленного изображения. Для того, чтобы все работало корректно внесем небольшие изменения в файлы `settings.py`, `urls.py`.

```
]
MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'
```

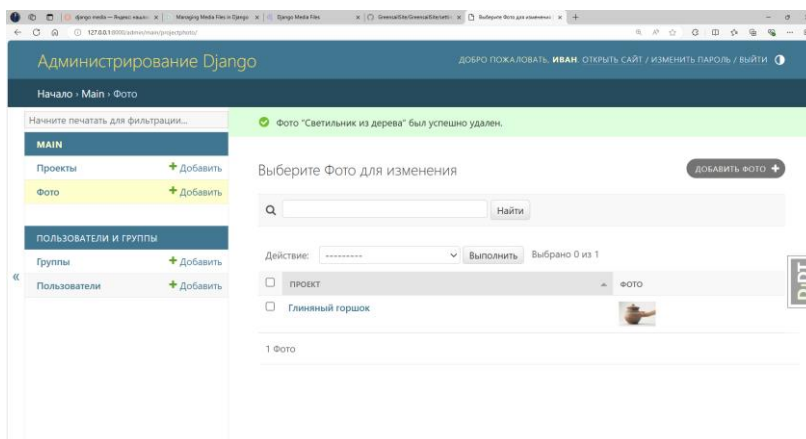
```
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import path, include
from django.conf import settings

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('main.urls')),
]

if settings.DEBUG:
    import debug_toolbar

    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls))
    ] + urlpatterns + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Готово. Посмотрим на результат.



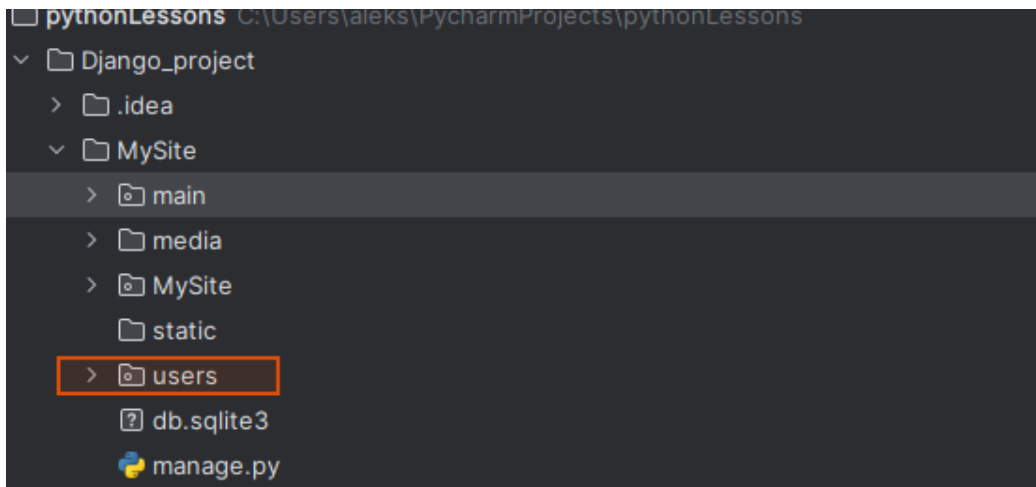
§9. Авторизация и регистрация пользователей.

После того, как мы научились работать с панелью администрации, хранить в базе данных медиафайлы и другую информацию – будет не лишним поговорить об авторизации и регистрации пользователей на сайте. Для того, чтобы вести учет пользователей, посещающих сайт – создадим новое приложение `users`, которое будет отвечать за работу с пользователями.

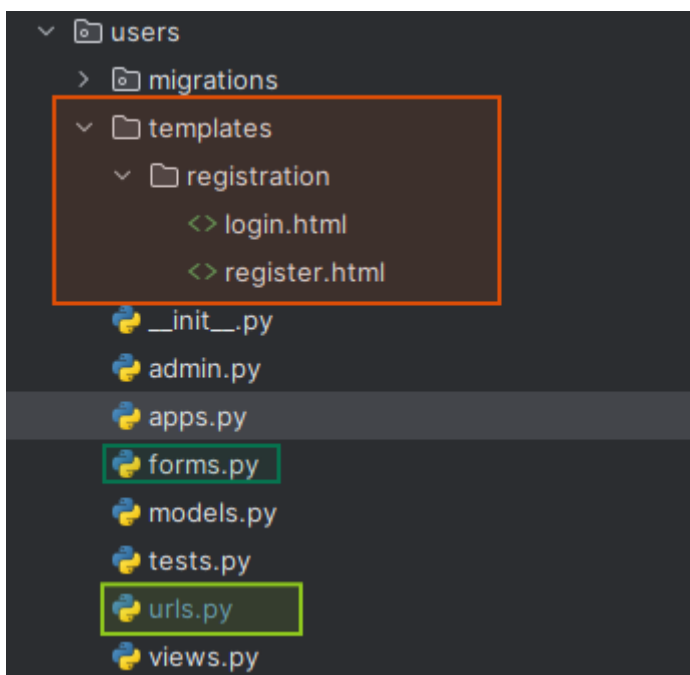
Для этого откроем терминал и введем уже знакомую нам команду:

`python.manage.py startapp users`

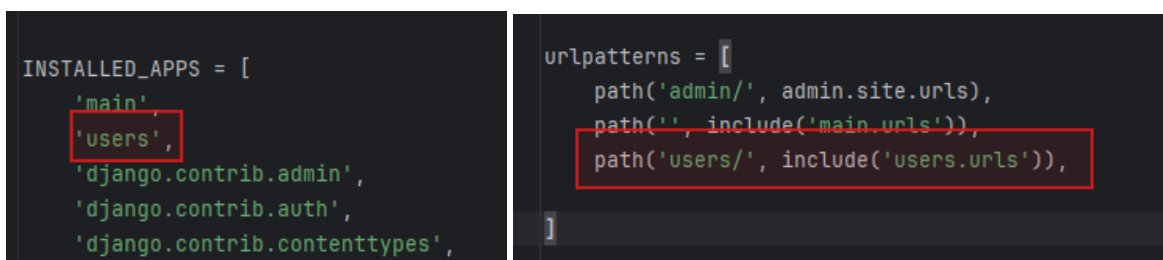
После выполнения в папке проекта у нас появится папка `users`, содержащая набор файлов, схожий с содержанием папки `main`



Для дальнейшей работы внесем изменения в содержание папки `users`: создадим подкаталог `templates`, а также файлы `forms.py`, `urls.py`. В подкаталоге `templates` создадим подкаталог `registration`, а в нем три шаблона: `login.html` и `register.html`.



Также необходимо зарегистрировать приложение `users` в файле `settings.py` и добавить url-адреса данного приложения в список url-адресов проекта в файле `urls.py`.



На этом подготовительные работы закончены и для начала сделаем возможной авторизацию пользователей. Для этого перейдем в файл `urls.py` в приложении `users` и добавим туда следующий код:

```
from django.urls import path, include
from . import views

app_name = 'users'
urlpatterns = [
    path('', include('django.contrib.auth.urls')),
]
```

```
from django.urls import path, include
from . import views
```

```
app_name = 'users'
urlpatterns = [
    path("", include('django.contrib.auth.urls')),
]
```

Для авторизации пользователей мы будем использовать стандартные инструменты, которые предлагает нам Django.

Разместим в файле `login.html` следующий код

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

{% block title %}Авторизация{% endblock %}

{% block content %}
    <form class="form" action="{% url 'users:login' %}" method="post">
        {% csrf_token %}
        {{form.as_p}}
        <button type="submit">Войти</button>
        <input type="hidden" name="next" value="{% url 'main:index' %}" />
    </form>
    <p>Еще нет аккаунта?</p>
    <a href="{% url 'users:register' %}">Регистрация</a>
{% endblock %}
```

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

```
{% block title %}Авторизация{% endblock %}
```

```
{% block content %}
    <form class="form" action="{% url 'users:login' %}" method="post">
```

```

{% csrf_token %}
{{ form.as_p }}
<button type="submit">Войти</button>
<input type="hidden" name="next" value="{% url 'main:index' %}" />
</form>
<p>Еще нет аккаунта?</p>
<a href="{% url 'users:register' %}">Регистрация</a>
{% endblock %}

```

Добавим в файл `urls.py` еще два url-адреса.

```

from django.urls import path, include
from . import views

app_name = 'users'
urlpatterns = [
    path('', include('django.contrib.auth.urls')),
    path('log_out/', views.logout_view, name='logout'),
    path('register/', views.register_view, name='register'),
]

```

```

from django.urls import path, include
from . import views

```

```

app_name = 'users'
urlpatterns = [
    path("", include('django.contrib.auth.urls')),
    path('log_out/', views.logout_view, name='logout'),
    path('register/', views.register_view, name='register'),
]

```

В файле views.py создадим заготовку функции представления авторизации и реализуем функцию представления logout_view

```
from django.contrib.auth import logout
from django.shortcuts import render, redirect

@aleks
def logout_view(request):
    logout(request)
    return redirect('main:index')

1 usage @aleks *
def register_view(request):
    return render(request, template_name='registration/register.html')
```

```
from django.contrib.auth import logout
from django.shortcuts import render, redirect
```

```
def logout_view(request):
    logout(request)
    return redirect('main:index')
```

```
def register_view(request):
    return render(request, 'registration/register.html')
```

Отлично. Теперь по url-адресу users/login можно попасть на страницу авторизации.

Имя пользователя:

Пароль:

Еще нет аккаунта?

[Регистрация](#)

Теперь организуем возможность регистрации пользователя на сайте. Для этого нам необходимо создать форму регистрации. Форма описывается в виде класса в файле forms.py. Давайте создадим ее.

```
from django.contrib.auth.forms import UserCreationForm
from django import forms
from django.contrib.auth.models import User

3 usages  ⤴ aleks

class UserRegisterForm(UserCreationForm):
    username = forms.CharField(max_length=250, label='Имя пользователя')
    first_name = forms.CharField(max_length=100, label='Имя')
    last_name = forms.CharField(max_length=100, label='Фамилия')
    email = forms.EmailField(max_length=100, label='Электронная почта')

⤴ aleks
def clean_email(self):
    email = self.cleaned_data['email']
    if User.objects.filter(email=email).exists():
        raise forms.ValidationError("Этот адрес электронной почты уже занят!")
    return email

⤴ aleks
class Meta:
    model = User
    fields = ['username', 'first_name', 'last_name', 'email',
              'password1', 'password2']
```

Разберемся, что же есть в данном коде.

Для начала необходимо импортировать некоторые модули и классы. Рассмотрим их более подробно и поговорим о том, для чего необходим каждый из них.

```
from django.contrib.auth.forms import UserCreationForm
from django import forms
from django.contrib.auth.models import User
```

Класс UserCreationForm, наследуемый нашим классом UserRegisterForm позволяет нам создать форму для регистрации на основе стандартной формы Django для создания пользователя. Модуль forms является аналогом модуля models и пригодится нам для определения типов данных полей формы. Класс User определяет стандартную модель пользователя в Django.

Теперь более подробно рассмотрим сам класс формы.

user_name, first_name, last_name и email являются полями формы и имеют свои типы данных, схожие с теми, что мы использовали при создании моделей. Аргумент max_length уже встречался ранее, а аргумент label позволяет переопределить подпись к полю.

Как и для моделей – для формы необходимо прописать метакласс, который содержит в себе информацию о модели, с которой связана данная форма (это необходимо для того, чтобы при отправке формы Django понимал, в какую таблицу в БД сохранить данные, которые ввел в форму пользователь), а также поля формы, которые будет предложено заполнить пользователю.

Наконец, поговорим о том, для чего необходим метод `clean_email`. Данный метод отвечает за проверку, не используется ли данный адрес электронной почты другим пользователем.

Таким образом полный код формы для регистрации имеет следующий вид:

```
from django.contrib.auth.forms import UserCreationForm
from django import forms
from django.contrib.auth.models import User

class UserRegisterForm(UserCreationForm):
    username = forms.CharField(max_length=250, label='Имя пользователя')
    first_name = forms.CharField(max_length=100, label='Имя')
    last_name = forms.CharField(max_length=100, label='Фамилия')
    email = forms.EmailField(max_length=100, label='Электронная почта')

    def clean_email(self):
        email = self.cleaned_data['email']
        if User.objects.filter(email=email).exists():
            raise forms.ValidationError("Этот адрес электронной почты уже занят!")
        return email

    class Meta:
        model = User
        fields = ['username', 'first_name', 'last_name', 'email',
                  'password1', 'password2']
```

Форма для регистрации готова и теперь можем перейти к написанию функции представления для отображения страницы регистрации. Для этого перейдем в файл `views.py` и внесем изменения в функцию `register_view`.

При отправке формы – сервер получает HTTP запрос POST, а вместе с ним данные от формы. Если пользователь только зашел на страницу регистрации – нам необходимо предложить ему пустую форму для заполнения, при этом отправка данных производиться не будет, а соответственно сервер получит запрос GET, означающий, что мы просим от него выдать какие-то данные, в нашем случае – страницу с пустой формой регистрации.

Таким образом нам необходимо проверить, какой запрос получает сервер и, в случае если этот запрос не POST – создать пустую форму регистрации и разместить ее на странице, а иначе проверить корректность заполнения формы и в случае, если форма прошла валидацию – сохранить данные пользователя в БД и авторизировать его на сайте.

Исходя из всего вышесказанного импортируем необходимые функции, классы и модули и напомним код функции представления для регистрации пользователей:

```
from django.contrib.auth import logout, login
from django.shortcuts import render, redirect
from .forms import UserRegisterForm

1 usage  ▲ aleks
def logout_view(request):
    logout(request)
    return redirect('main:index')

1 usage  ▲ aleks
def register_view(request):
    if request.method != 'POST':
        form = UserRegisterForm()
    else:
        form = UserRegisterForm(data=request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)
            return redirect('main:index')
    context = {'form': form}
    return render(request, template_name='registration/register.html', context)
```

```
def register_view(request):
    if request.method != 'POST':
        form = UserRegisterForm()
    else:
        form = UserRegisterForm(data=request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)
            return redirect('main:index')
    context = {'form': form}
    return render(request, 'registration/register.html', context)
```

Осталось еще немного. Теперь необходимо написать шаблон страницы регистрации.

Разместим следующий код в файле register.html

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

{% block title %}Регистрация{% endblock %}

{% block content %}
    <form action="{% url 'users:register' %}" method="post">
        {% csrf_token %}
        {{form.as_p}}
        <button type="submit">Зарегистрироваться</button>
        <input type="hidden" name="next" value="{% url 'main:index' %}">
    </form>
{% endblock %}
```

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

```
{% block title %}Регистрация{% endblock %}
```

```
{% block content %}
```

```
<form action="{% url 'users:register' %}" method="post">
```

```
    {% csrf_token %}
```

```
    {{ form.as_p }}
```

```
<button type="submit">Зарегистрироваться</button>
```

```
<input type="hidden" name="next" value="{% url 'main:index' %}">
```

```
</form>
```

```
{% endblock %}
```

Важное замечание. В разметке страниц авторизации и регистрации можно заметить строки

```
{% csrf_token %}
```

```
{{ form.as_p }}
```

Первая отвечает за передачу так называемого csrf-токена, который необходим для безопасности, вторая позволяет выгрузить на страницу форму с помощью шаблонизатора.

Отлично. Теперь у нас есть возможность регистрировать пользователей на сайте.

Имя пользователя:

Имя:

Фамилия:

Электронная почта:

Пароль:

- Пароль не должен быть слишком похож на другую вашу личную информацию.
- Ваш пароль должен содержать как минимум 8 символов.
- Пароль не должен быть слишком простым и распространенным.
- Пароль не может состоять только из цифр.

Подтверждение пароля: Для подтверждения введите, пожалуйста, пароль ещё раз.

Аутентификация по паролю:

☒ Разрешена
☐ Запрещена

Может ли пользователь аутентифицироваться по паролю. Если эта возможность выключена, пользователь всё ещё может аутентифицироваться иным способом, например, Single Sign-On или LDAP, если они сконфигурированы и разрешены.

Теперь внесем некоторые изменения в базовый шаблон. Для этого перейдем в файл base.html и добавим туда следующий код

```
<body>
  <a href="{% url 'main:index' %}">Главная</a>-
  <a href="{% url 'main:about_me' %}">Обо мне</a>-
  <a href="{% url 'main:projects' %}">Мои работы</a>-
  <a href="{% url 'main:order' %}">Заказать</a>-
  {% if user.is_authenticated %}
    <span>
      <strong>Здравствуйте, {{user.username}}!</strong>
    </span>
    {% if user.is_superuser %}
      <a href="{% url 'admin:index' %}">Управление</a>-
    {% endif %}
    <a href="{% url 'users:logout' %}">Выйти</a>
  {% else %}
    <a href="{% url 'users:login' %}">Войти</a>
  {% endif %}
{% block content %}{% endblock %}
</body>
</html>
```

```

<a href="{% url 'main:order' %}">Заказать</a>-
{% if user.is_authenticated %}
    <span>
        <strong>Здравствуйте, {{user.username}}!</strong>
    </span>
    {% if user.is_superuser %}
        <a href="{% url 'admin:index' %}">Управление</a>-
    {% endif %}
    <a href="{% url 'users:logout' %}">Выйти</a>
{% else %}
    <a href="{% url 'users:login' %}">Войти</a>
{% endif %}

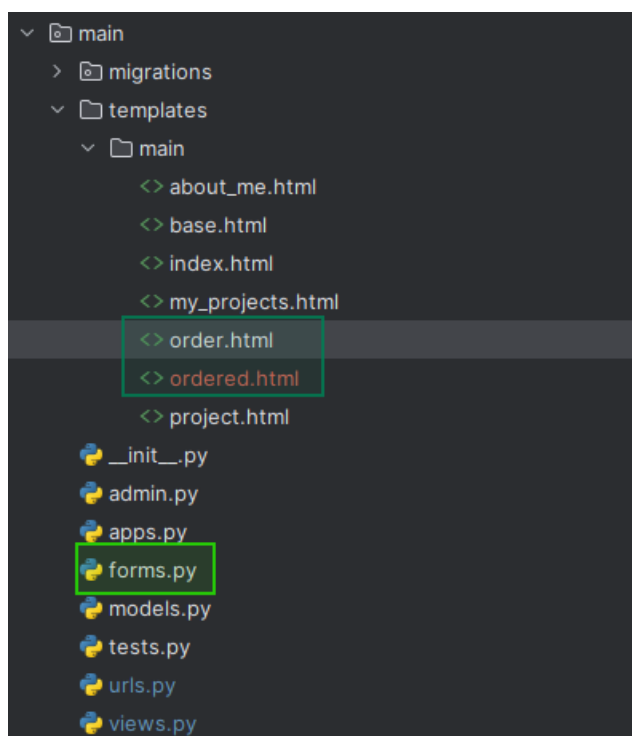
```

Теперь в панели навигации у нас будет отображаться приветствие и кнопка «выйти», если пользователь авторизован или кнопка «войти», а также раздел «заказать», которым мы займемся в следующей главе. Если пользователь является администратором сайта, то в панели управления у него будет доступна кнопка «управление», позволяющая перейти в админ-панель для управления сайтом.

§10. Оформление заказов. Работа с формами в Django

В предыдущей главе мы начали знакомство с формами в Django и узнали, что форма, как и модель задается в виде класса. Для создания класса формы регистрации мы использовали готовое решение на основании класса UserCreationForm, которое нам предлагает Django. В этой главе мы научимся создавать свои собственные формы на примере формы оформления заказа.

Прежде чем начать работать с формой – проведем некоторые подготовительные работы в приложении main, а именно создадим файл forms.py и шаблоны order.html и ordered.html



Формы в Django создаются на основании моделей, а значит нам необходимо создать модель, отвечающую за заказы. Для этого перейдем в файл models.py и разместим там код класса модели Order

```
3 usages  ▲ aleks
class Order(models.Model):
    theme = models.CharField(max_length=250, verbose_name='Тема')
    user = models.ForeignKey(User, on_delete=models.CASCADE, verbose_name='Пользователь')
    email = models.EmailField(max_length=100, verbose_name='Электронная почта')
    phone_number = models.CharField(verbose_name='Номер телефона')
    address = models.CharField(max_length=300, verbose_name='Адрес доставки')
    description = models.TextField(verbose_name='Описание заказа')
    photo = models.ImageField(upload_to='media/orderPhotos', verbose_name='Фото', null=True)

    ▲ aleks
    def __str__(self):
        return self.theme

    ▲ aleks
    class Meta:
        verbose_name = 'Заказ'
        verbose_name_plural = 'Заказы'
```

```
class Order(models.Model):
    theme = models.CharField(max_length=250, verbose_name='Тема')
    user = models.ForeignKey(User, on_delete=models.CASCADE,
verbose_name='Пользователь')
    email = models.EmailField(max_length=100, verbose_name='Электронная почта')
    phone_number = models.CharField(verbose_name='Номер телефона')
    address = models.CharField(max_length=300, verbose_name='Адрес доставки')
    description = models.TextField(verbose_name='Описание заказа')
    photo = models.ImageField(upload_to='media/orderPhotos', verbose_name='Фото',
null=True)

    def __str__(self):
        return self.theme

    class Meta:
        verbose_name = 'Заказ'
        verbose_name_plural = 'Заказы'
```

Модель Order, как вы можете заметить связана через внешний ключ с моделью пользователя, а поэтому нам необходимо импортировать данную модель:

```
user = models.ForeignKey(User, on_delete=models.CASCADE, verbose_name='Пользователь')

from django.contrib.auth.models import User
from django.db import models
```

Теперь выполним миграции с помощью команд в терминале:

python manage.py makemigrations

```
python manage.py migrate
```

Готово. Наша модель синхронизирована с базой данных, и мы можем переходить к написанию класса формы OrderForm, который будет располагаться в файле forms.py.

Для начала импортируем модели и модуль forms

```
from django import forms
from .models import Order
```

Теперь определим сам класс формы, подобно тому, как мы делали для формы регистрации, но теперь будем строить класс на основе класса forms.ModelForm

```
class OrderForm(forms.ModelForm):
    theme = forms.CharField(max_length=250, label='Тема')
    email = forms.EmailField(max_length=100, label='Электронная почта')
    phone_number = forms.CharField(max_length=250, label='Номер телефона')
    address = forms.CharField(max_length=300, label='Адрес доставки')
    description = forms.Textarea()
    photo = forms.ImageField(label='Фото')

    class Meta:
        model = Order
        fields = ['theme', 'email', 'phone_number', 'address', 'description', 'photo']
```

В итоге код в файле forms.py будет следующим:

```
from django import forms
from .models import Order
```

```
class OrderForm(forms.ModelForm):
    theme = forms.CharField(max_length=250, label='Тема')
    email = forms.EmailField(max_length=100, label='Электронная почта')
    phone_number = forms.CharField(max_length=250, label='Номер телефона')
    address = forms.CharField(max_length=300, label='Адрес доставки')
    description = forms.Textarea()
    photo = forms.ImageField(label='Фото')

    class Meta:
        model = Order
        fields = ['theme', 'email', 'phone_number', 'address', 'description', 'photo']
```

Форма написана и теперь мы можем реализовать функцию представления для страницы оформления заказа – order_view.

Проверка запроса производится аналогично тому, как мы делали в предыдущей главе

```
def order_view(request):
    if request.method != 'POST':
        form = OrderForm()
    else:
```

Теперь поговорим о блоке else. Дело в том, что в данном случае форма связана с моделью, которая через внешний ключ связана с пользователем, а потому перед отправкой формы – нам нужно дополнительно записать в форму, какой пользователь ее отправляет, поскольку пользователю не предлагается самостоятельно заполнить это поле.

Реализуется это посредством создания переменной, в которую записывается, форма, но не отправляется, а далее в поле user заносится информация о пользователе, от которого пришел запрос на отправку формы, после чего форма сохраняется и отправляется, а данные из нее сохраняются в БД.

```
new_order = form.save(commit=False)
new_order.user = request.user
form.save()
```

Таким образом полный код функции представления order_view имеет следующий вид:

```
def order_view(request):
    if request.method != 'POST':
        form = OrderForm()
    else:
        form = OrderForm(data=request.POST, files=request.FILES)
        if form.is_valid():
            new_order = form.save(commit=False)
            new_order.user = request.user
            form.save()
            return redirect('main:ordered')

    context = {'form': form}
    return render(request, 'main/order.html', context)
```

```
def order_view(request):
    if request.method != 'POST':
        form = OrderForm()
    else:
        form = OrderForm(data=request.POST, files=request.FILES)
        if form.is_valid():
            new_order = form.save(commit=False)
            new_order.user = request.user
            form.save()
            return redirect('main:ordered')

    context = {'form': form}
    return render(request, 'main/order.html', context)
```


Также реализуем небольшую функцию представления для страницы, на которую попадает пользователь после оформления заказа.

```
def ordered(request):  
    return render(request, template_name='main/ordered.html')
```

```
def ordered(request):  
    return render(request, 'main/ordered.html')
```

Также было бы не лишним позаботиться о том, чтобы эти страницы могли видеть только авторизованные пользователи. Для этого будем использовать декоратор `login_required`, который необходимо импортировать и добавить к функциям представления.

```
from django.contrib.auth.decorators import login_required
```

Не забудьте также импортировать формы:

```
from .forms import *
```

```
from django.contrib.auth.decorators import login_required  
from django.shortcuts import render, redirect  
from .models import *  
from .forms import *
```

Далее навешиваем на нужные функции представления сам декоратор и передаем ему в аргументы `login_url`. Таким образом, если неавторизованный пользователь попытается зайти или получить доступ к страницам `order` или `ordered` – он не сможет сделать это и будет автоматически переправлен по url-адресу `users/login`, где ему будет предложено авторизоваться или зарегистрироваться. Использование данного декоратора позволяет избежать ряда ошибок, повышает безопасность, а самое главное делает невозможной ситуацию, в которой неавторизованный или незарегистрированный на сайте пользователь мог бы оформить заказ, что недопустимо ввиду, как неудобства для администратора сайта, который занимается изготовлением и отправкой заказов пользователям, так и ввиду работы приложения в целом, поскольку далее мы реализуем отправку электронного письма пользователю в случае оформления заказа, а для того, чтобы это сделать – необходимо знать данную информацию о пользователе, соответственно пользователь должен быть зарегистрирован и авторизован на сайте.

```

@login_required(login_url='users:login')
def order_view(request):
    if request.method != 'POST':
        form = OrderForm()
    else:
        form = OrderForm(data=request.POST, files=request.FILES)
        if form.is_valid():
            new_order = form.save(commit=False)
            new_order.user = request.user
            form.save()
            return redirect('main:ordered')

    context = {'form': form}
    return render(request, template_name='main/order.html', context)

1 usage new *
@login_required(login_url='users:login')
def ordered(request):
    return render(request, template_name='main/ordered.html')

```

Осталось лишь сопоставить url-адреса. Сделаем это в файле urls.py для приложения main.

```

urlpatterns = [
    path('', views.index, name='index'),
    path('about_me', views.about_me, name='about_me'),
    path('my_projects', views.my_projects, name='projects'),
    path('my_projects/<slug:project_slug>', views.project_view, name='project'),
    path('order/', views.order_view, name='order'),
    path('ordered/', views.ordered, name='ordered')
]

```

```

path('order/', views.order_view, name='order'),
path('ordered/', views.ordered, name='ordered')

```

Теперь перейдем к написанию разметки для шаблонов order.html и ordered.html:

order.html

```

{% extends 'main/base.html' %}

{% block title %}Оформление заказа{% endblock %}

{% block content %}
    <form action="{% url 'main:order' %}" method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{form.as_p}}
        <button type="submit">Подтвердить</button>
    </form>
    <input type="hidden" name="next" value="{% url 'main:index' %}">
{% endblock %}

```

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

```
{% block title %}Оформление заказа{% endblock %}
```

```
{% block content %}
```

```
    <form action="{% url 'main:order' %}" method="post" enctype="multipart/form-data">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Подтвердить</button>
    </form>
    <input type="hidden" name="next" value="{% url 'main:index' %}">
```

```
{% endblock %}
```

Атрибут формы `enctype="multipart/form-data"` необходим для возможности загрузки изображений.

ordered.html

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

{% block title %}Заказ оформлен{% endblock %}

{% block content %}
    <h1>Спасибо за ваш заказ!</h1>
    <h2>Администратор сайта скоро свяжется с вами для уточнения деталей.</h2>
    <a href="{% url 'main:index' %}" type="button">Вернуться на сайт</a>
{% endblock %}
```

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

```
{% block title %}Заказ оформлен{% endblock %}
```

```
{% block content %}
```

```
    <h1>Спасибо за ваш заказ!</h1>
    <h2>Администратор сайта скоро свяжется с вами для уточнения деталей.</h2>
    <a href="{% url 'main:index' %}" type="button">Вернуться на сайт</a>
```

```
{% endblock %}
```

Готово. Теперь можем посмотреть на результат:

[Главная](#)- [Обо мне](#)- [Мои работы](#)- [Заказать](#)- **Здравствуйте, admin1!** [Управление](#)- [Выйти](#)

Тема:

Электронная почта:

Номер телефона:

Адрес доставки:

Описание:

Фото: Не выбран ни один файл

[Главная](#)- [Обо мне](#)- [Мои работы](#)- [Заказать](#)- **Здравствуйте, admin1!** [Управление](#)- [Выйти](#)

Тема:

Электронная почта:

Номер телефона:

Адрес доставки:

Описание:

Фото: 1_027_785.jpg

[Главная](#)- [Обо мне](#)- [Мои работы](#)- [Заказать](#)- **Здравствуйте, admin1!** [Управление](#)- [Выйти](#)

Спасибо за ваш заказ!

Администратор сайта скоро свяжется с вами для уточнения деталей.

[Вернуться на сайт](#)

Теперь давайте зарегистрируем нашу модель на сайте администрации и напишем для нее класс OrderAdmin. Для этого перейдем в файл admin.py в приложении main.

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'theme', 'user', 'get_html_photo']
    list_display_links = ['theme']
    search_fields = ['theme']
    readonly_fields = ('get_html_photo', )

    1 usage  ▲ aleks
    def get_html_photo(self, object):
        if object.photo:
            return mark_safe(f'')

    get_html_photo.short_description = 'Фото'
```

```
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'theme', 'user', 'get_html_photo']
    list_display_links = ['theme']
    search_fields = ['theme']
    readonly_fields = ('get_html_photo', )

    def get_html_photo(self, object):
        if object.photo:
            return mark_safe(f'')

    get_html_photo.short_description = 'Фото'
```

```
admin.site.register(models.Order, OrderAdmin)
```

```
admin.site.register(models.Order, OrderAdmin)
```

Готово. Теперь можем перейти в панель администрации и убедиться, что заказы, оформленные через форму действительно отображаются и сохраняются в базу данных корректно.

Выберите Заказ для изменения

ДОБАВИТЬ ЗАКАЗ +

Q


Найти

Действие:





▼

Выполнить

Выбрано 0 из 1

<input type="checkbox"/>	ID	ТЕМА	ПОЛЬЗОВАТЕЛЬ	ФОТО
<input type="checkbox"/>	6	Глиняный горшок	admin1	

1 Заказ

Тема:	<input type="text" value="Глиняный горшок"/>
Пользователь:	<input type="text" value="admin1"/>   
Электронная почта:	<input type="text" value="alekseev.i260303@gmail.com"/>
Номер телефона:	<input type="text" value="89867281288"/>
Адрес доставки:	<input type="text" value="Green city"/>
Описание заказа:	<div>Хочу горшок из глины. С узорами!</div>
Фото:	<div>На данный момент: media/orderPhotos/1_027_785_kvUzvNO.jpg Изменить: <input type="button" value="Выбор файла"/> Не выбран ни один файл</div>
Фото:	

§11. Отправка электронных писем в Django

Было бы удобно, если при оформлении пользователем заказа на сайте – администратор получал письмо о том, что оформлен новый заказ и подробности по заказу, а пользователь в свою очередь письмо с подтверждением о том, что заказ оформлен. Давайте реализуем это.

Первое, что необходимо сделать – внести изменения в файл settings.py

```
MEDIA_URL = '/media/'
MEDIA_ROOT = BASE_DIR / 'media'

# Default primary key field type
# https://docs.djangoproject.com/en/5.1/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'

INTERNAL_IPS = [
    '127.0.0.1',
]

EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = 587
EMAIL_HOST_USER = "alekseev.i260303@gmail.com"
EMAIL_HOST_PASSWORD = "vcum hrpf ibcj lkml"
EMAIL_USE_TLS = True
EMAIL_USE_SSL = False
```

Останавливаться на этом не будем, более подробно рекомендую почитать на данном ресурсе: [Настройка отправки писем email в Django для mail, яндекс, gmail | VIVAZZI](#) или в официальной документации Django: [Sending email | Django documentation | Django \(djangoproject.com\)](#)

Теперь займемся непосредственно отправкой писем. Делать это необходимо в случае успешной отправки формы перед тем, как перенаправить пользователя на страницу о

том, что заказ оформлен. Для этого внесем изменения в функцию представления `order_view`.

Отправка писем производится с помощью функции `send_mail`. Давайте импортируем ее:

```
from django.contrib.auth.decorators import login_required
from django.shortcuts import render, redirect
from .models import *
from .forms import *
from django.conf import settings
from django.core.mail import send_mail
```

```
from django.conf import settings
from django.core.mail import send_mail
```

Теперь давайте попробуем отправить письмо на электронную почту пользователя, который оформил заказ. Для этого добавим следующий код в функцию представления:

```
1 usage  aleks *
@login_required(login_url='users:login')
def order_view(request):
    if request.method != 'POST':
        form = OrderForm()
    else:
        form = OrderForm(data=request.POST, files=request.FILES)
        if form.is_valid():
            new_order = form.save(commit=False)
            new_order.user = request.user
            form.save()
            send_mail(subject='Заказ оформлен', message=f'Добрый день, {request.user.username}!\n'
                    f'Ваш заказ {new_order} успешно оформлен. Администратор сайта скоро свяжется '
                    f'с вами!', settings.EMAIL_HOST_USER, recipient_list=[new_order.email])
            return redirect('main:ordered')
```

`send_email` принимает следующие аргументы:

`send_mail('Тема', 'Тело письма', settings.EMAIL_HOST_USER, ['to@example.com']),`

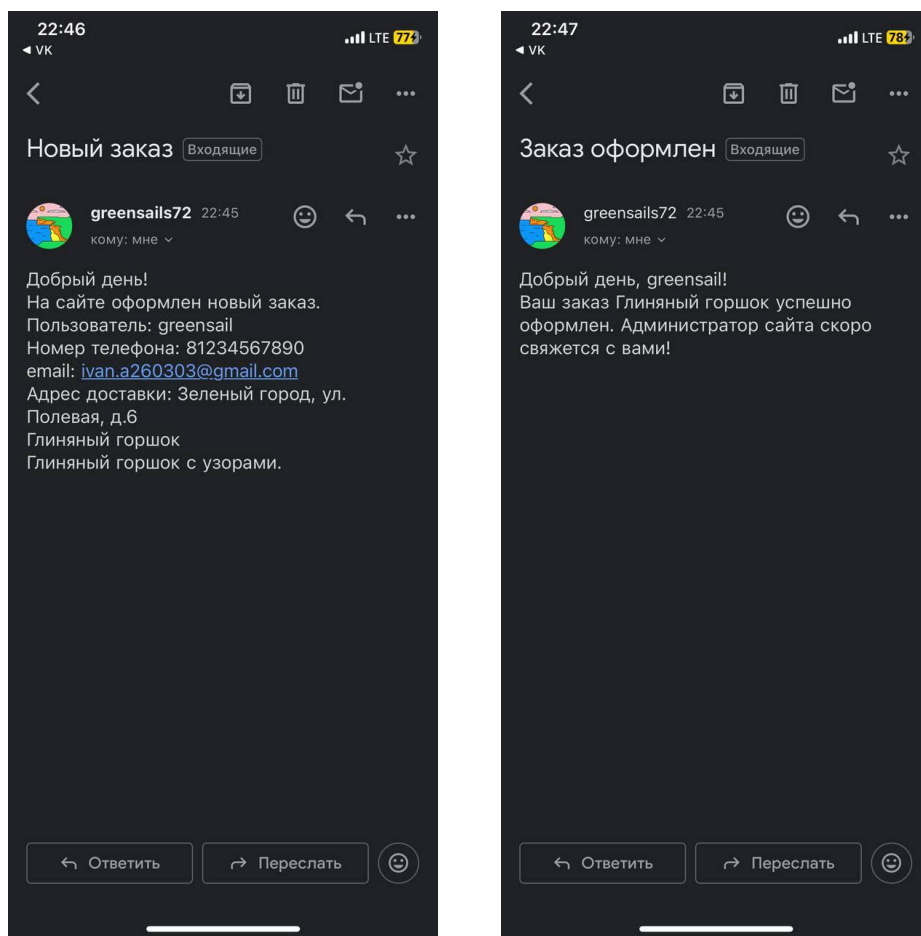
где `'to@example.com'` – адрес электронной почты получателя.

Теперь давайте добавим отправку письма администратору с уведомлением о том, что оформлен новый заказ:

```
new_order = form.save(commit=False)
new_order.user = request.user
form.save()
send_mail(subject='Заказ оформлен', message=f'Добрый день, {request.user.username}!\n'
        f'Ваш заказ {new_order} успешно оформлен. Администратор сайта скоро свяжется '
        f'с вами!', settings.EMAIL_HOST_USER, recipient_list=[new_order.email])
send_mail(subject='Новый заказ', message=f'Добрый день!\nНа сайте оформлен новый заказ.'
        f'\nПользователь: {request.user.username}\n'
        f'Номер телефона: {new_order.phone_number}\n'
        f'email: {new_order.email}\n'
        f'Адрес доставки: {new_order.address}\n'
        f'{new_order.theme}\n'
        f'{new_order.description}', settings.EMAIL_HOST_USER,
        recipient_list=['alekseev.i260303@gmail.com'])
return redirect('main:ordered')
```

В качестве почты получателя необходимо указать электронную почту, на которую администратору будут приходить уведомления.

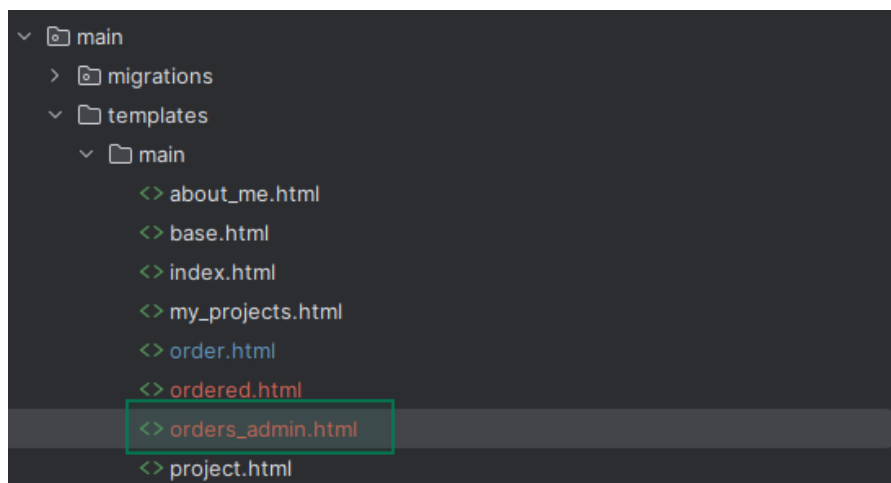
Отлично. Мы научились отправлять электронные письма в Django.



§12. Страница с заказами пользователей

Для удобства работы реализуем страницу, на которой будут размещаться все заказы, оформленные на данный момент, и информация о них, которая будет доступна только администраторам сайта. Это будет сделано для того, чтобы администратор мог удобно просматривать оформленные заказы.

Для этого создадим в приложении main новый шаблон orders_admin.html



Прежде чем перейти к сопоставлению url-адресов и написанию функции представления – внесем изменения в базовый шаблон base.html


```

<body>
  <a href="{% url 'main:index' %}">Главная</a>-
  <a href="{% url 'main:about_me' %}">Обо мне</a>-
  <a href="{% url 'main:projects' %}">Мои работы</a>-|
  <a href="{% url 'main:order' %}">Заказать</a>-
  {% if user.is_authenticated %}
    <span>
      <strong>Здравствуйте, {{user.username}}!</strong>
    </span>
    {% if user.is_superuser %}
      <a href="{% url 'main:orders_admin' %}">Актуальные заказы</a>-
      <a href="{% url 'admin:index' %}">Управление</a>-
    {% endif %}
    <a href="{% url 'users:logout' %}">Выйти</a>
  {% else %}
    <a href="{% url 'users:login' %}">Войти</a>
  {% endif %}

  {% block content %}{% endblock %}
</body>
/html>

```

Теперь сопоставим url-адреса

```

app_name = 'main'
urlpatterns = [
    path('', views.index, name='index'),
    path('about_me', views.about_me, name='about_me'),
    path('my_projects', views.my_projects, name='projects'),
    path('my_projects/<slug:project_slug>', views.project_view, name='project'),
    path('order/', views.order_view, name='order'),
    path('ordered/', views.ordered, name='ordered'),
    path('actual_orders', views.actual_orders_view, name='orders_admin'),
]

```

path('actual_orders', views.actual_orders_view, name='orders_admin'),

Реализуем функцию представления actual_orders_view

```

@login_required(login_url='users:login')
def actual_orders_view(request):
    orders = Order.objects.prefetch_related('user').order_by('-id')
    conntext = {'actual_orders': orders}
    return render(request, template_name='main/orders_admin.html', conntext)

```

@login_required(login_url='users:login')

```

def actual_orders_view(request):
    orders = Order.objects.prefetch_related('user').order_by('-id')
    conntext = {'actual_orders': orders}
    return render(request, 'main/orders_admin.html', conntext)

```

Готово. Теперь можем перейти к написанию кода шаблона orders_admin.html

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

{% block title %}Актуальные заказы{% endblock %}

{% block content %}
    {% for order in actual_orders %}
        <div class="order">
            <h1>{{order}}</h1>
            <h2>Заказчик: {{order.user.username}}</h2>
            <h2>ФИО: {{order.user.first_name}} {{order.user.last_name}}</h2>
            <h2>Телефон: {{order.phone_number}}</h2>
            <h2>Адрес электронной почты: {{order.email}}</h2>
            <h2>Пожелания пользователя:<br>{{order.description}}</h2>
            <h2>Фото:</h2>
            
            <h2>Адрес доставки: {{order.address}}</h2>
        </div>
    {% empty %}
        <h1>На данный момент нет актуальных заказов</h1>
    {% endfor %}
{% endblock %}
```

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

```
{% block title %}Актуальные заказы{% endblock %}
```

```
{% block content %}
```

```
    {% for order in actual_orders %}
```

```
        <div class="order">
```

```
            <h1>{{order}}</h1>
```

```
            <h2>Заказчик: {{order.user.username}}</h2>
```

```
            <h2>ФИО: {{order.user.first_name}} {{order.user.last_name}}</h2>
```

```
            <h2>Телефон: {{order.phone_number}}</h2>
```

```
            <h2>Адрес электронной почты: {{order.email}}</h2>
```

```
            <h2>Пожелания пользователя:<br>{{order.description}}</h2>
```

```
            <h2>Фото:</h2>
```

```
            
```

```
            <h2>Адрес доставки: {{order.address}}</h2>
```

```
        </div>
```

```
    {% empty %}
```

```
        <h1>На данный момент нет актуальных заказов</h1>
```

```
    {% endfor %}
```

```
{% endblock %}
```

Посмотрим на полученный результат

[Главная](#) - [Обо мне](#) - [Мои работы](#) - [Заказать](#) - [Здравствуйте, admin!](#) [Актуальные заказы](#) - [Управление](#) - [Выйти](#)

Глиняный горшок

Заказчик: greensail

ФИО: green sail

Телефон: 81234567890

Адрес электронной почты: ivan.a260303@gmail.com

Пожелания пользователя:

Глиняный горшок с узорами.

Фото:



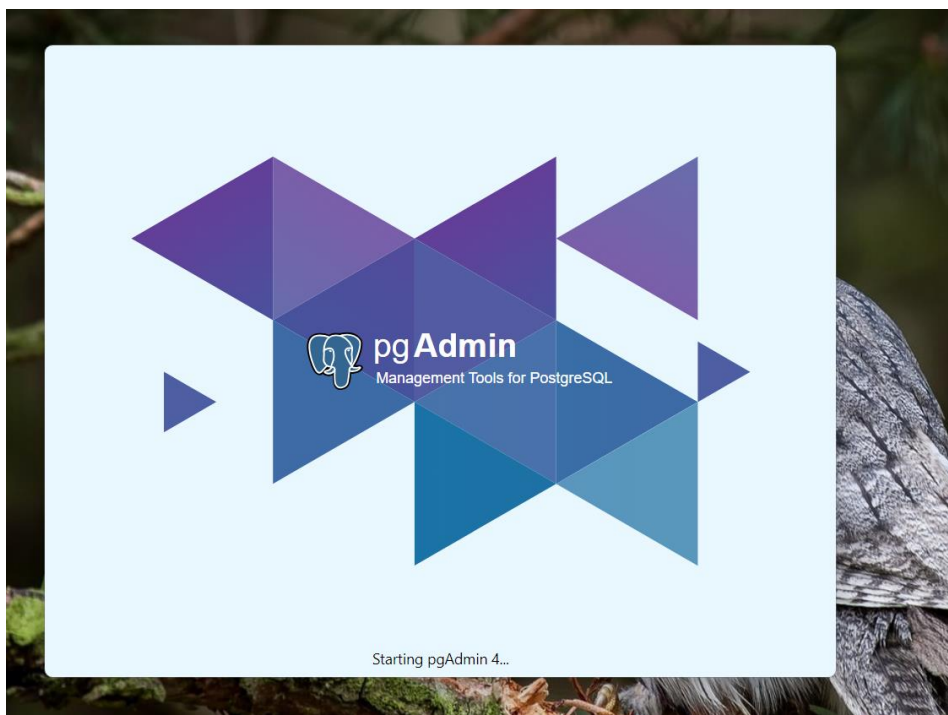
Адрес доставки: Зеленый город, ул. Полевая, д.6

§13. Перенос базы данных

На этом наша работа с серверной частью приложения подходит к концу, но прежде чем приступить к оформлению клиентской части приложения – перенесем имеющиеся в БД данные с sqlite3 на PostgreSQL.

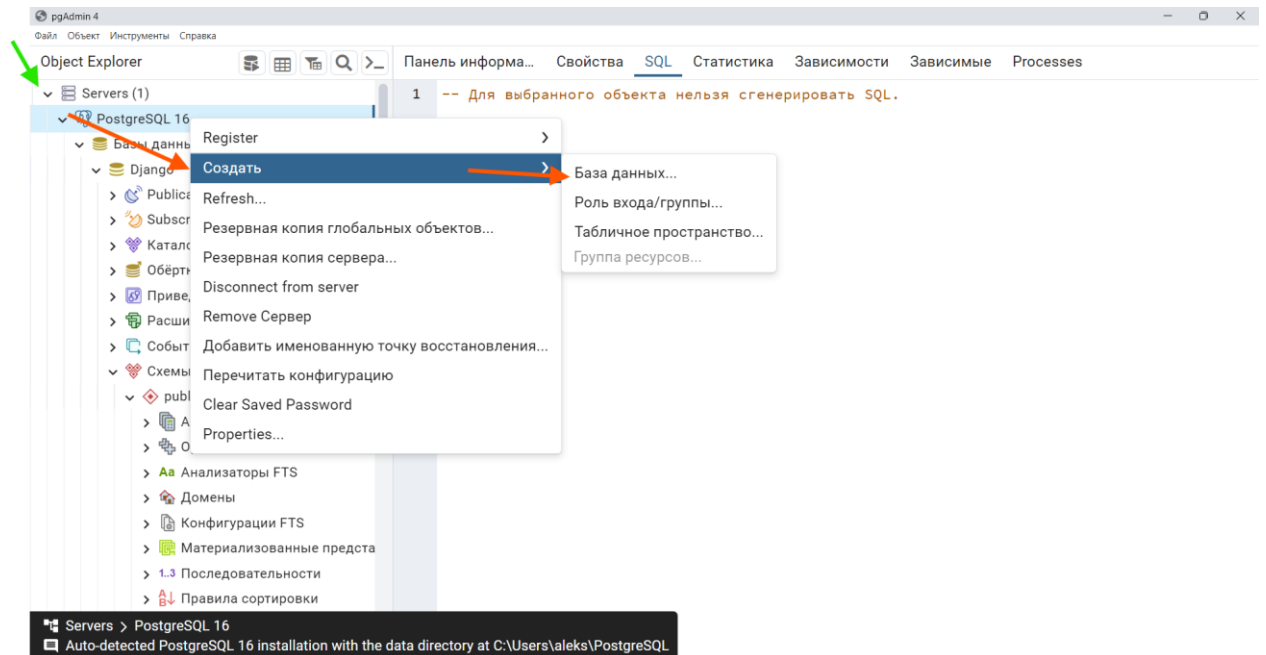
Для начала необходимо установить PostgreSQL. Вы можете сделать это, загрузив установщик с официального сайта: [PostgreSQL: Windows installers](#)

После установки проведем некоторые подготовительные работы. Для этого необходимо запустить приложение pgAdmin.

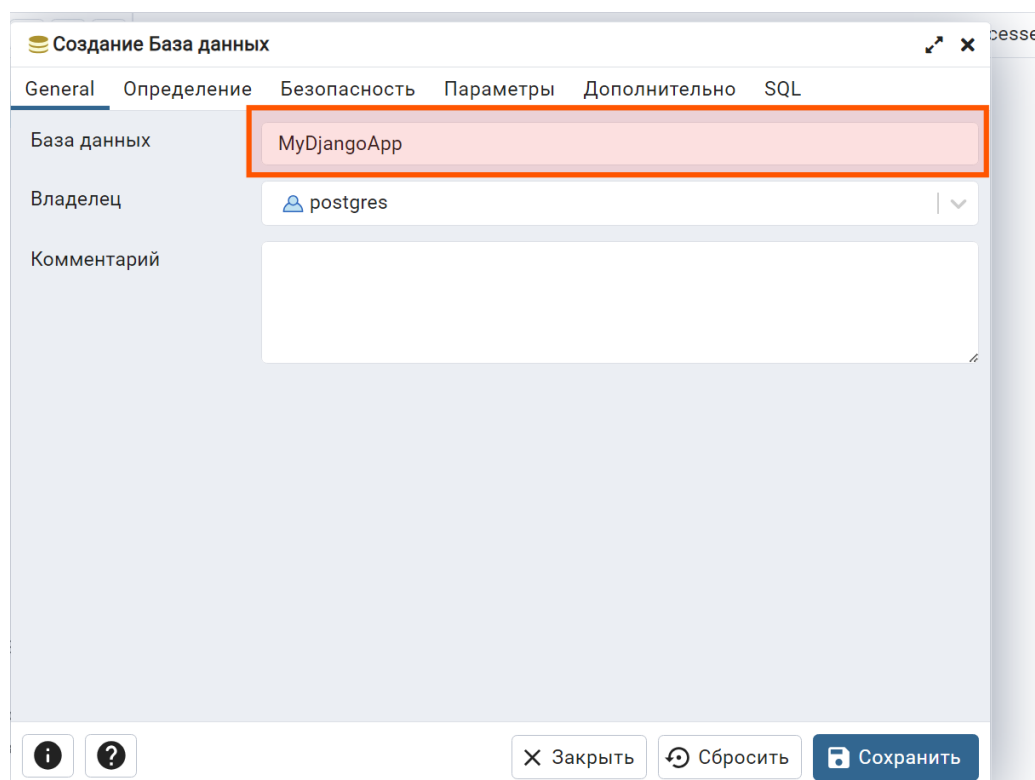


При первом входе – вам будет предложено создать аккаунт. При создании не забудьте записать учетные данные. Позже они вам пригодятся.

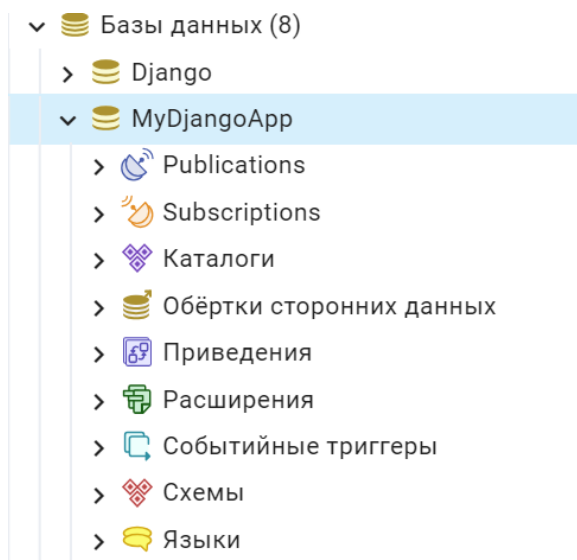
Далее создадим новую базу данных:



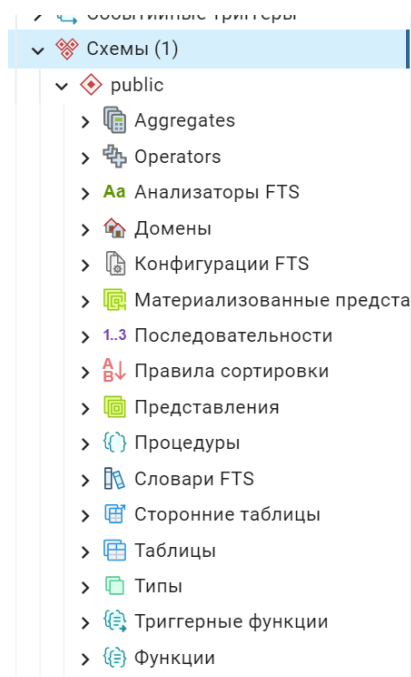
Указываем название и сохраняем



Теперь найдем нашу базу данных в списке и посмотрим, что для нее у нас есть



Для того, чтобы посмотреть, какие таблицы есть в базе данных откроем раздел «схемы»



На данный момент у нас нет таблиц, но вскоре мы их добавим.

Теперь, прежде чем вносить изменения в наш проект – создадим дамп существующей базы данных. Для этого перейдем в терминал и выполним следующую команду:

```
python -Xutf8 manage.py dumpdata --indent=2 --exclude auth.permission --exclude contenttypes -o db.json
```

После выполнения у нас в проекте должен появиться файл db.json, содержащий дамп существующей базы данных.



Внесем некоторые изменения в файл settings.py, а именно подключим новую базу данных, используя данные, которые вы установили при регистрации.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'MyDjangoApp',
        'USER': 'postgres',
        'PASSWORD': '12345678',
        'HOST': 'localhost'
    }
}
```

Теперь выполним миграции с помощью команды:

`python manage.py migrate`

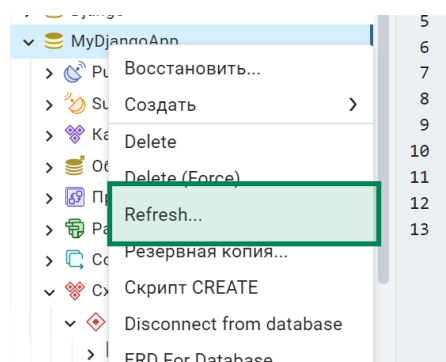
```
(venv) PS C:\Users\aleks\PycharmProjects\pythonLessons\Django_project\MySite> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, main, sessions, users
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying main.0009_order_orderphotos... OK
  Applying main.0010_alter_orderphotos_photo... OK
  Applying main.0011_order_photo_delete_orderphotos... OK
  Applying main.0012_alter_order_options_alter_order_phone_number... OK
  Applying main.0013_alter_order_phone_number... OK
  Applying sessions.0001_initial... OK
  Applying users.0001_initial... OK
  Applying users.0002_alter_useravatar_avatar... OK
  Applying users.0003_alter_useravatar_avatar... OK
  Applying users.0004_delete_useravatar... OK
(venv) PS C:\Users\aleks\PycharmProjects\pythonLessons\Django_project\MySite>
```

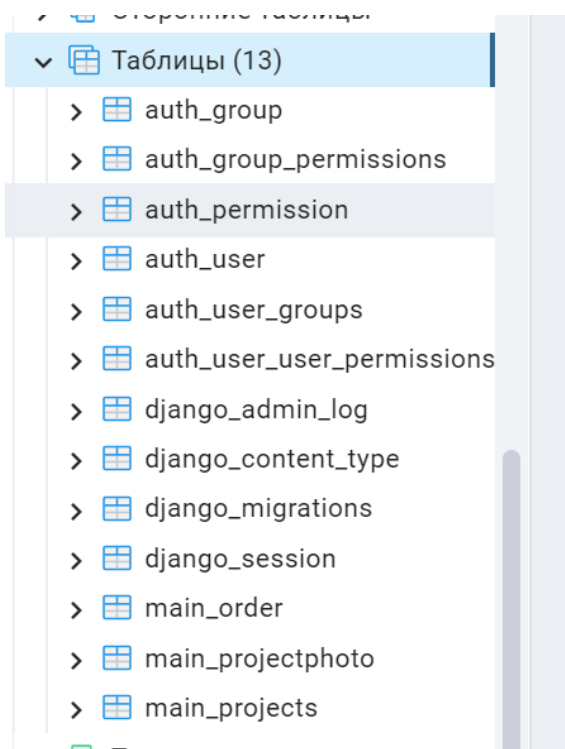
Осталось лишь записать данные из дампа в новую базу данных, выполнив команду:

`python manage.py loaddata db.json`

```
(venv) PS C:\Users\aleks\PycharmProjects\pythonLessons\Django_project\MySite> python manage.py loaddata db.json
Installed 19 object(s) from 1 fixture(s)
```

Вернемся в pgAdmin и посмотрим на раздел «таблицы», предварительно обновив БД.





В качестве примера посмотрим содержимое таблицы `auth_user`. Для этого откроем запросник и введем следующий SQL-запрос:

Скриншот интерфейса административной панели базы данных. В центре экрана отображен SQL-запрос: `SELECT * FROM auth_user;`. В нижней части экрана отображены результаты запроса в виде таблицы. Таблица имеет 7 столбцов: `id`, `password`, `last_login`, `is_superuser`, `username`, `first_name` и `last_name`. В таблице 5 записей.

	id	password	last_login	is_superuser	username	first_name	last_name
1	teU52d9cpQm...	2024-09-05 17:30:32.63+03	true	aleks			
2	DScBlZbX3HA...	2024-09-05 17:30:07.052+03	false	admin	Иван	Алексеев	
3	hJlOfKAi9k=	2024-10-17 22:55:39.438+03	true	admin1			
4	jcSVINFAAQfU...	2024-10-17 13:49:52.554+03	false	admin123	Иван	Алексеев	
5	hhlMOnRdOkRV...	2024-10-17 22:29:41.524+03	false	greensail	green	sail	

`SELECT * FROM auth_user;`

Отлично. Наши данные перенесены, и на этом мы заканчиваем работу с серверной частью приложения и переходим к работе с клиентской частью, подробнее с которой мы познакомимся в главе 2.

Глава 2. Клиентская часть приложения.