

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:

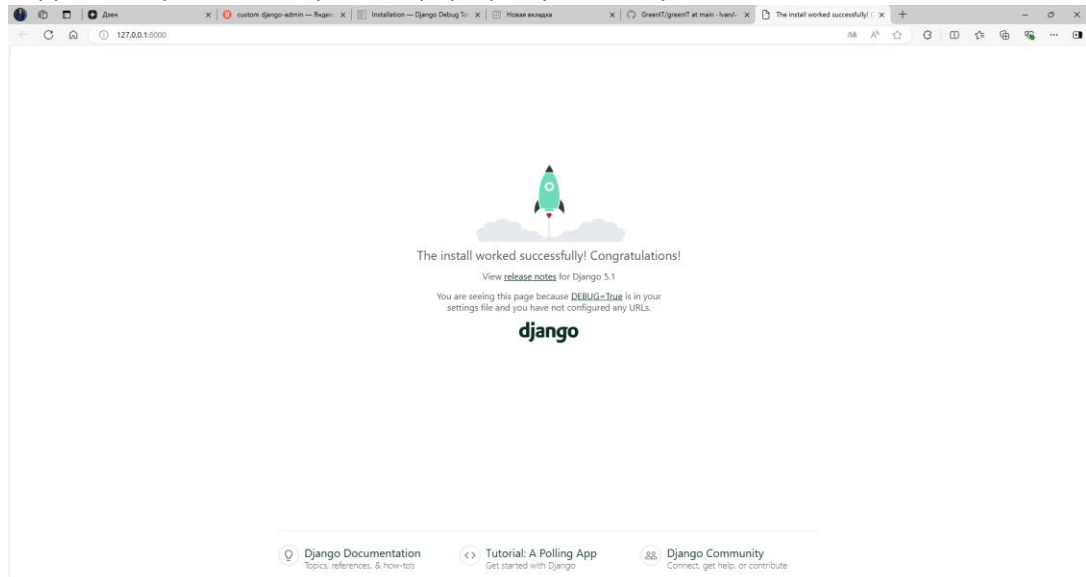
```
django-admin startproject <Имя проекта>
```

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

1.6 Выполняем команды:

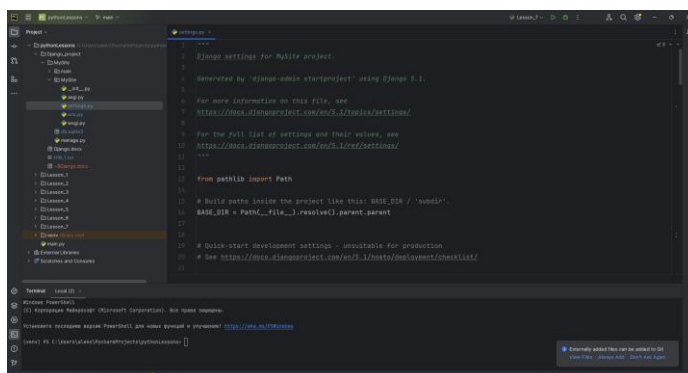
```
python manage.py migrate
python manage.py runserver
```

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».

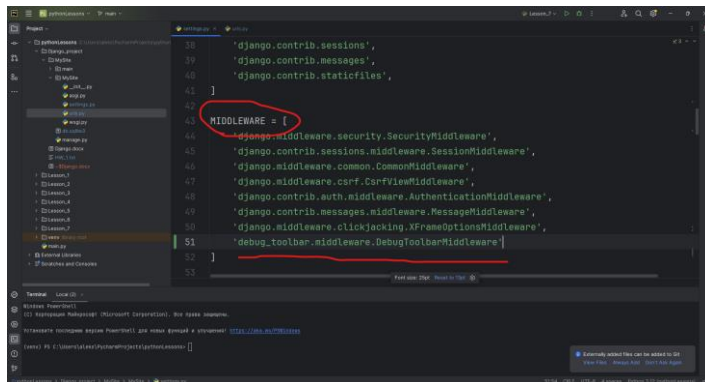
Далее установим зависимости для статических файлов и медиа(для хранения изображений, видео и т.п в БД).

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

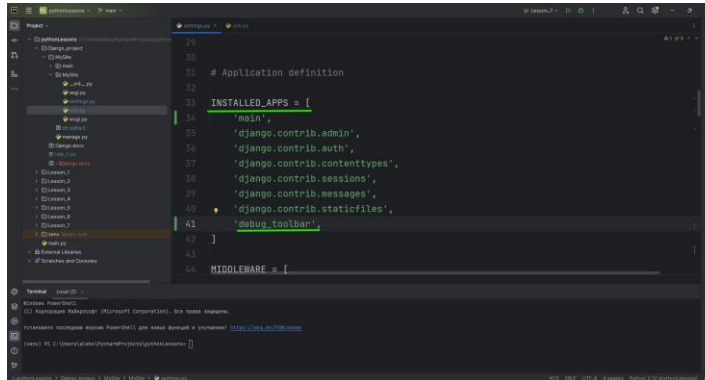
```
STATICFILES_DIRS = [
    BASE_DIR / "static",
]
```

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

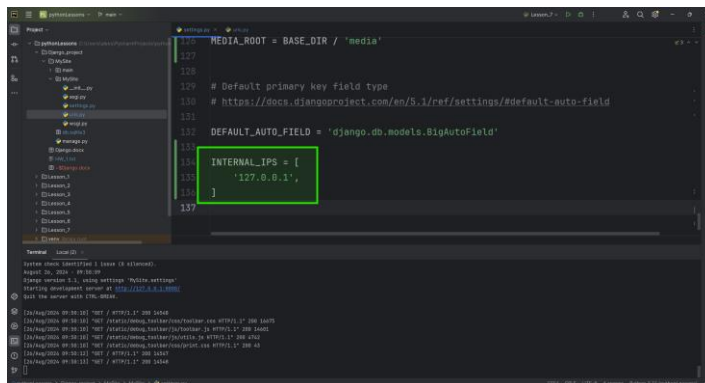
Теперь заранее позаботимся об установке отладочной панели Django, о которой будем вести разговор в дальнейшем. Для этого необходимо добавить `'debug_toolbar.middleware.DebugToolbarMiddleware'` в список `MIDDLEWARE`.



Далее добавляем «debug_toolbar» в список `INSTALLED_APPS`.



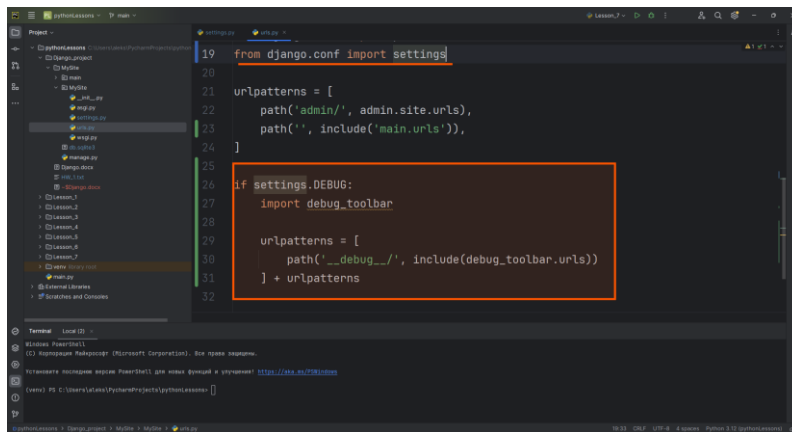
В конце файла добавляем следующий код



```
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.

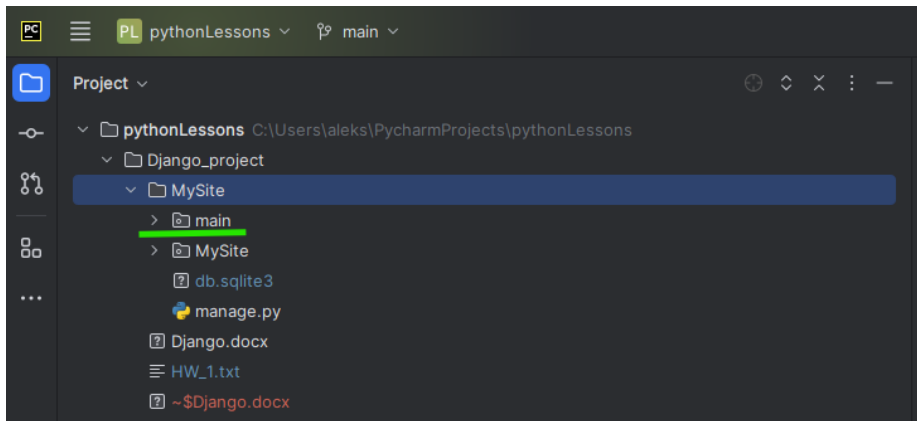
3. Создание первого приложения. Модель MTV.

2.1 Для дальнейшей работы над проектом необходимо создать приложение

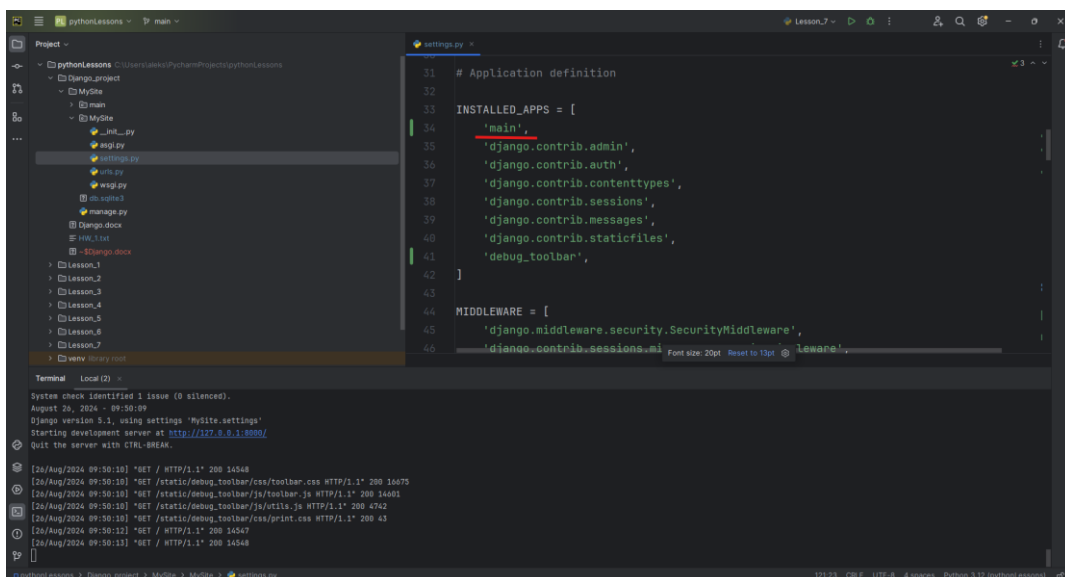
Для этого с использованием команд в терминале, описанных в блоке 1 переходим в папку проекта и выполняем следующую команду:

```
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.

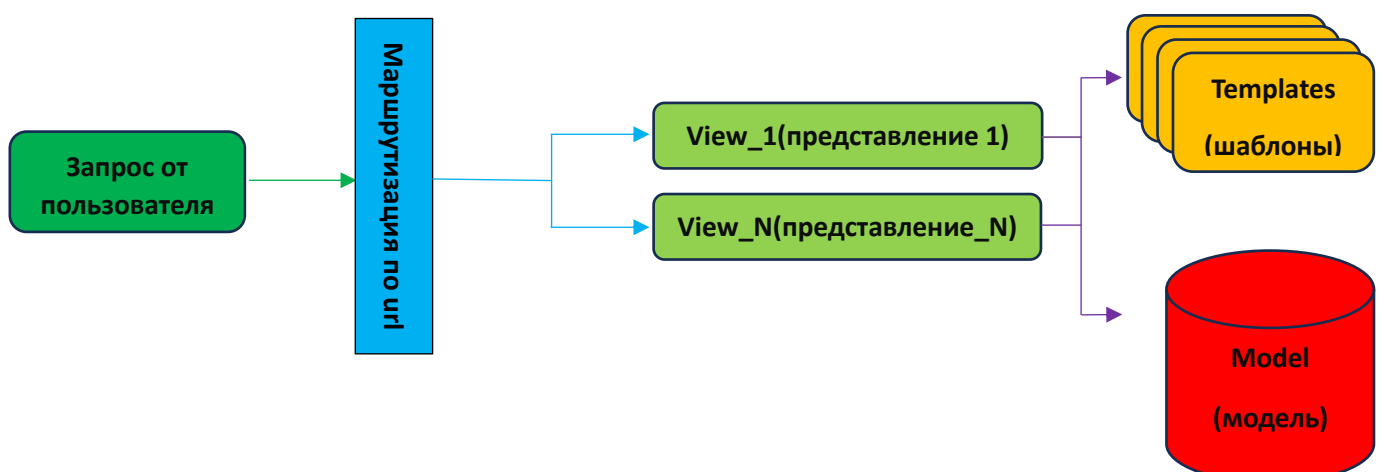
Django работает по модели MTV – Models Templates Views.

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):
    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

```
<!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>
```

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.

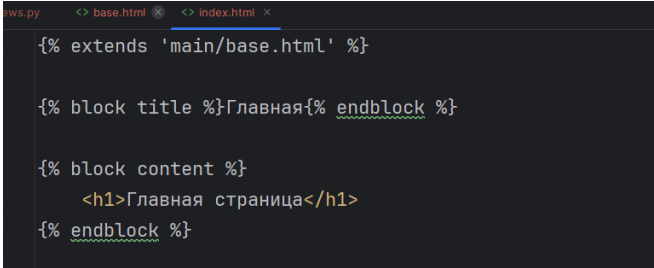
```
{% block title %}Главная{% endblock %}
```

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

```
    <h1>Главная страница</h1>
```

```
{% endblock %}
```

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



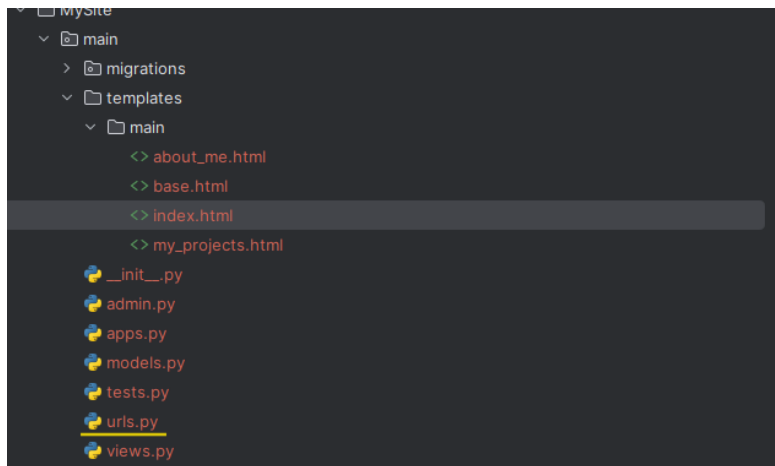
```
{% extends 'main/base.html' %}

{% 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
```

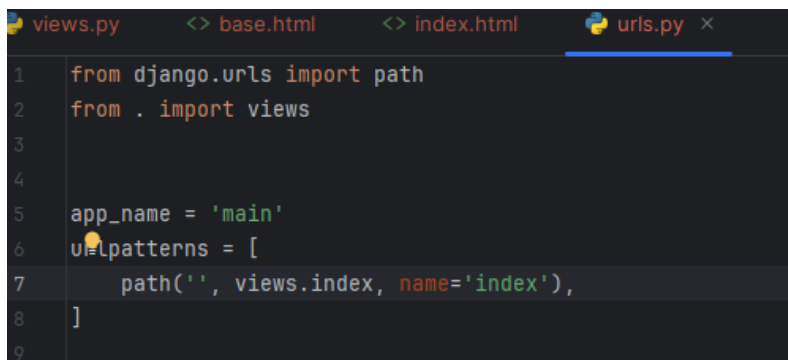
Теперь для дальнейшего удобства при обращении к url адресам данного приложения – укажем его название в переменной `app_name`

```
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



```
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 написать шаблоны – 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

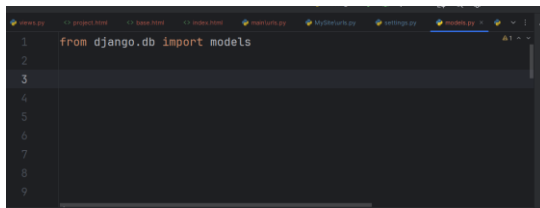
Пока что нам будет достаточно данного набора тегов для дальнейшей работы. В случае необходимости в дальнейшем мы изучим и другие теги.

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:

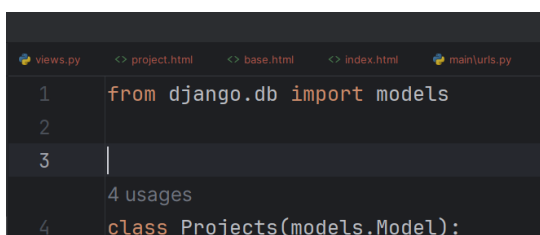
project_name(CharField) – название работы

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

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:

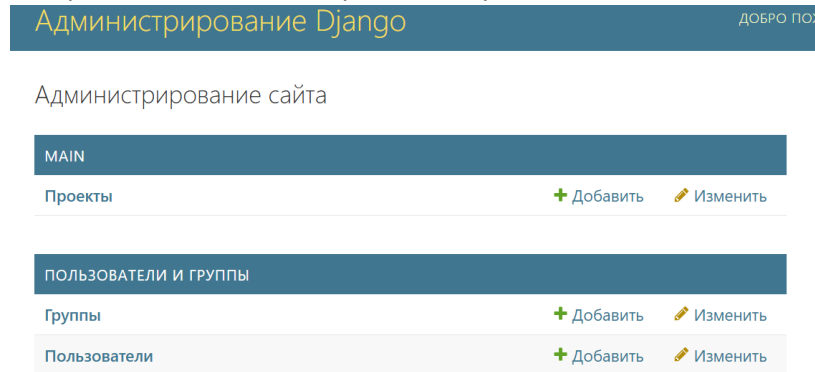
```
admin.site.register(<Имя модели>)
```

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:

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, template_name='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.

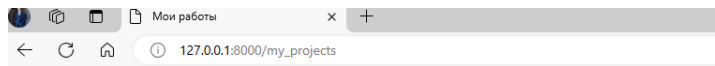
```
<ul>
  {% for project in projects %}
    <li>
      <h4>Date added: {{project.date_added}}</h4>
      <a href="">{{project}}</a>

    </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:

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

    context = {'project': project}

    return render(request, '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 %}

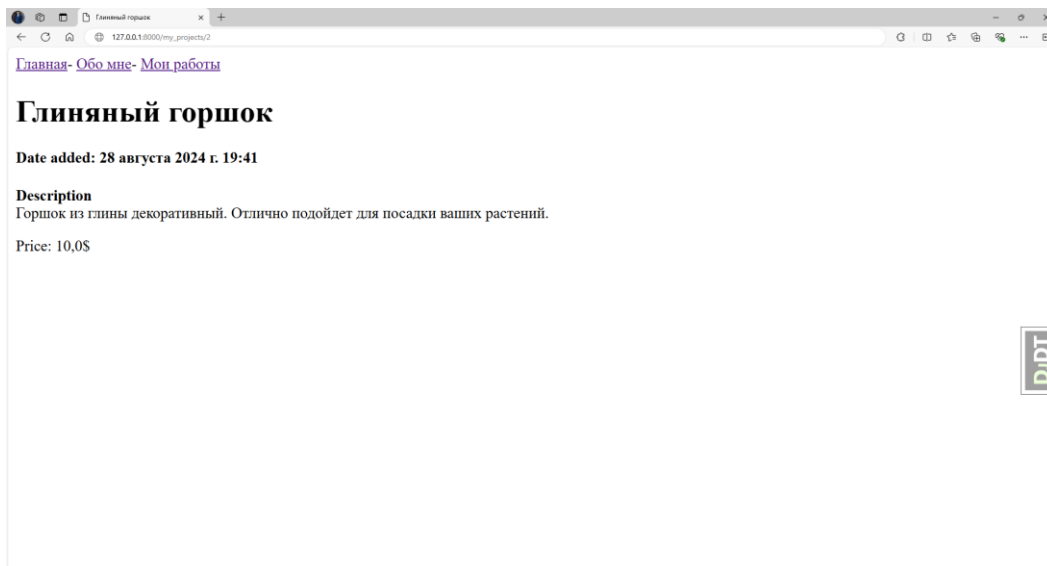
{% block content %}
  <h1>{{project}}</h1>
  <h4>Date added: {{project.date_added}}</h4>
  <p>
    <strong>Description</strong>
    <br>
    {{project.description}}
  </p>
  <p>Price: {{project.price}}$</p>
{% endblock %}
```

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  {% endblock %}
20
21
```

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



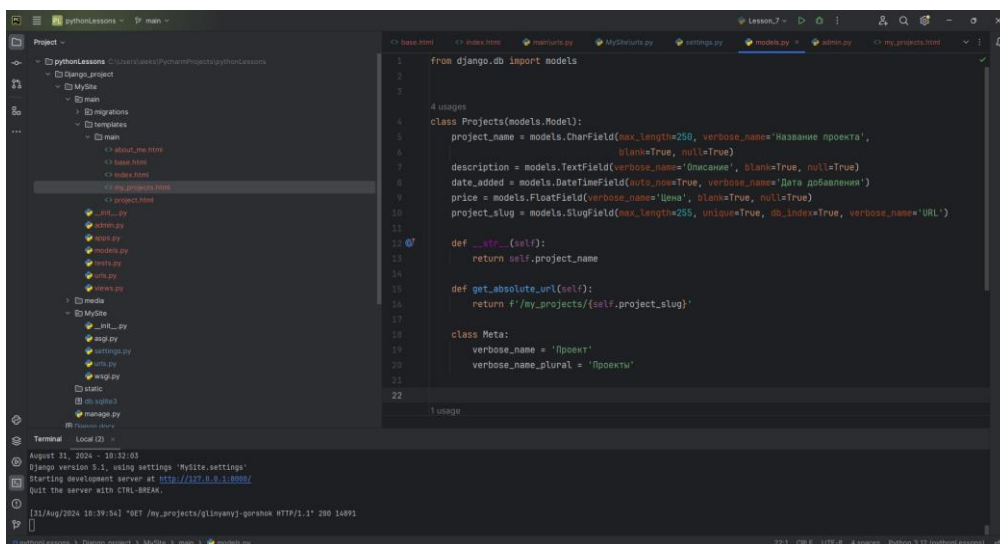
Обращаться в url по id не очень удобно и безопасно. Поэтому сделаем так, чтобы у каждой динамически-изменяемой страницы в адресной строке было свое короткое имя, которое называется слаг. Для добавления слагов необходимо добавить в модель соответствующее поле.

```
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 слаг.

```
def project_view(request, project_slug):
    project = Projects.objects.get(project_slug=project_slug)

    context = {'project': project}

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

```
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
13         </li>
14     {% empty %}
15         <li>
16             <h3>На данный момент нет проектов</h3>
17         </li>
18     {% endfor %}
19
20 </ul>
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

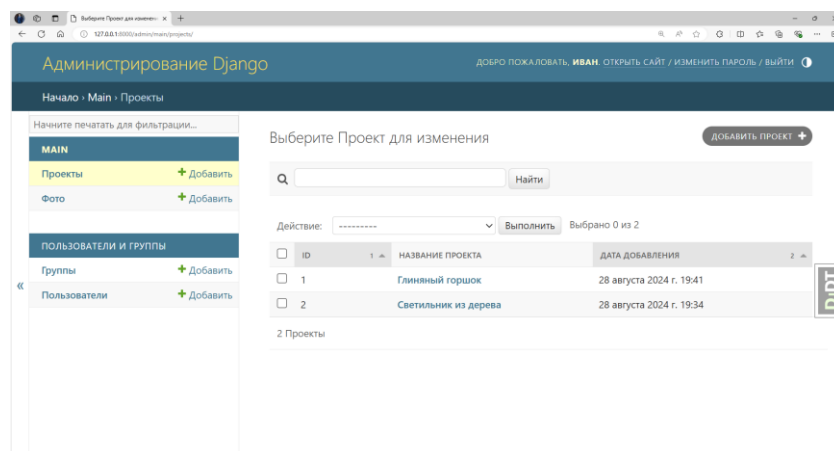
Список `search_fields` определяет поля, по которым можно вести поиск

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'
```

```
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.

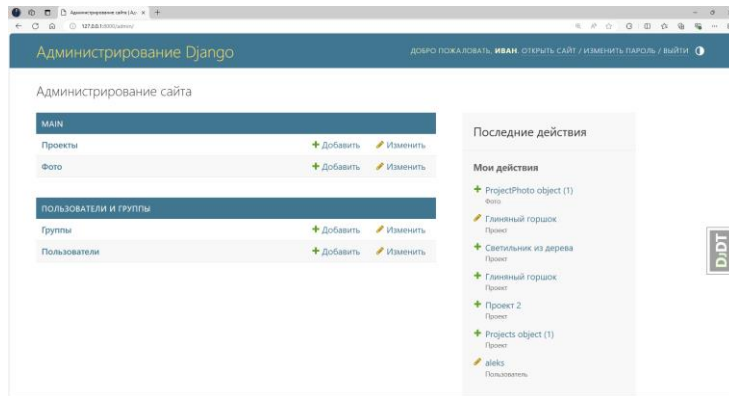
Атрибут `upload_to` определяет, куда будут сохраняться добавляемые файлы.

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.



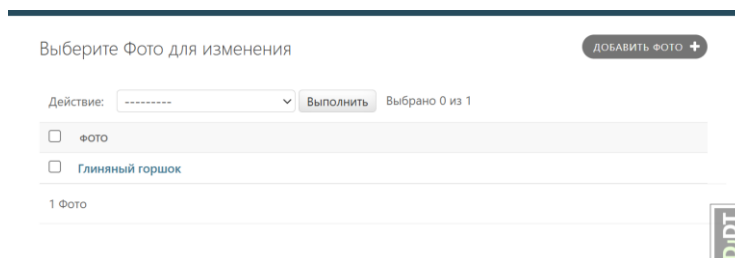
Добавим также метод `__str__`, который будет возвращать имя проекта, к которому относятся добавленные фотографии.

```
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',)

    1 usage
    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.

