Методическое пособие:

Знакомство с фреймворком Django

Автор: Грушевский Юрий Викторович

Контакты: jurapro@yandex.ru

Оглавление

[**1. Введение**](#_860qtefy4d9v) **3**

[Для кого предназначено данное методическое пособие](#_fllkl791hghn) 3

[Необходимое программное обеспечение](#_5iht9ouf8czo) 3

[Необходимые начальные знания](#_8pofxy8nubd) 3

[**2. Начало работы**](#_qk931d4mzws7) **4**

[План выполнения работы](#_dlfng9b3hxpt) 4

[Создание нового проекта и базового приложения](#_602aaex2xva) 5

[Создание основного приложения main](#_x1lo00wg5zdd) 7

[Создание базового шаблона](#_cdtvvqhhtpe3) 8

[Главная веб-страница](#_ww1y9hwxjbow) 12

[Дополнительные веб-страницы](#_d7mswj5ntls6) 15

[**3. Работа с пользователями**](#_9h17m6naxk2l) **17**

[Модель пользователя](#_ybg25r8vm0e4) 17

[Страницы входа, профиля и выхода](#_f3unyzfmhtf8) 19

[Страница входа](#_ptqt3u1j0mwg) 19

[Страница профиля](#_95qj4jeqtta0) 21

[Страница выхода](#_y2d2z7aia2v3) 22

[Страницы редактирования личных данных пользователя](#_x516ifx6lviq) 23

[Страница правки основных сведений](#_4xxb81prgv58) 23

[Страница изменения пароля](#_t45jysqixb6p) 26

[Страницы регистрации и активации пользователей](#_cq8i8cqgei3m) 27

[Форма для занесения сведений о новом пользователе](#_ua4q1sea0y9h) 28

[Контроллеры и шаблоны для регистрации пользователя](#_3ndks9317jtf) 29

[Средства для отправки писем с требованием активации](#_r7hx1jfhi3us) 31

[Страницы активации пользователя](#_xwyk12djzz6l) 33

[Страница удаления пользователя](#_vxspiy1ezwkc) 36

[Инструменты для администрирования пользователей](#_d6qgke81hsrz) 37

[**4. Рубрики**](#_35wah5792f48) **40**

[Модели рубрик](#_8h86ng56lpyj) 40

[Базовая модель рубрик](#_qrrf7v4onuz7) 40

[Модель надрубрик](#_hpk3w9und5u7) 41

[Модель подрубрик](#_o74zp45vwylv) 42

[Инструменты для администрирования рубрик](#_q9txocs3zkeo) 43

[Вывод рубрик в панели навигации](#_yzrlrnp8seox) 45

[**5. Объявления**](#_ub1kz11cq82j) **48**

[Обработка загруженных файлов](#_iw3nqoldword) 48

[Модели объявлений и дополнительных иллюстраций](#_306s46hwfx1v) 50

[Модель самих объявлений](#_js36zflyh5k) 50

[Модель дополнительных иллюстраций](#_6z2ros3glsvj) 52

[Реализация удаления объявлений в модели пользователя](#_5p1770wmi60n) 53

[Инструменты для администрирования объявлений](#_8txls3n6tn7w) 53

[Вывод объявлений](#_orkp5si06pr8) 54

[Форма поиска и контроллер списка объявлений](#_v4uhr1hr356a) 54

[Реализация корректного возврата](#_xsfilcborlde) 56

[Шаблон страницы списка объявлений](#_cnc4napeh2ku) 57

[Вывод сведений о выбранном изображении](#_ik60o78sc8wj) 59

[Вывод последний 10 объявлений на главной странице](#_xhklhipgdbq8) 62

[Работа с объявлениями](#_czv3y0hwlo4q) 63

[Вывод объявлений, оставленных текущим пользователем](#_ia1fqgwcpdoc) 63

[Добавление, правка и удаление объявлений](#_l8csk3vlna6l) 63

[**6. Комментарии**](#_7v2cpgyndexu) **68**

[Подготовка к выводу CAPTCHA](#_az6vyncdi4h3) 68

[Модель комментария](#_q46m8w21ldbx) 68

[Вывод и добавление комментариев](#_o3t7mpfw9fr0) 70

[Отправка уведомлений о появлении новых комментариев](#_j5mgpd8gfjb0) 72

[**7. Веб-служба REST**](#_mbeqgbw49kij) **74**

[Предварительная настройка](#_nozjorscx6yh) 74

[Подготовка к разработке веб-службы](#_dpwfct8frvsf) 74

[Список объявлений](#_oye832m0qje2) 75

[Сведения о выбранном объявлений](#_hkprc9o7qk3b) 77

[Вывод и добавление комментариев](#_4didkahug0yu) 78

# 

# 1. Введение

## Для кого предназначено данное методическое пособие

Данное пособие не ставит перед собой цель показать все возможности фреймворка Django, его можно позиционировать скорее как методический материал, с которым можно ознакомится, чтобы составить общее впечатление об этом фреймворке. Он по возможности максимально избавлен от сложных решений в угоду простоте и скорости разработки. Подойдет для самостоятельного изучения и использования как основы для организации учебных практик в учебных заведениях.

## Необходимое программное обеспечение

В качестве интерпретатора был использован Python 3.6, который можно загрузить на официальном сайте - <https://www.python.org/downloads/>.

В роли среды разработки была выбрана программа JetBrains PyCharm, так как это один из самых популярных IDE для данного фреймворка на текущий момент. Скачать ее можно с официального сайта - <https://www.jetbrains.com/pycharm/download/>. Доступно две редакции: Professional и Community. Вторая бесплатная и ее возможностей вполне достаточно для выполнения примеров из данного пособия.

## Необходимые начальные знания

Предполагается, что читатель знаком с основами синтаксиса языка Python, основами HTML и CSS, а также основой работы в командной строке (терминале). Кроме того, большим подспорьем будет знакомство с PyCharm и протоколом HTTP.

# 2. Начало работы

## План выполнения работы

Данное методическое пособие представляет собой практическое упражнение - разработку полнофункционального веб-сайта “Электронная доска объявлений”.

Наша электронная доска объявлений позволит зарегистрированным пользователям

публиковать объявления о продаже чего-либо.

Объявления будут располагаться в рубриках, причем структура рубрик будет иметь два уровня иерархии: на первом уровне расположатся рубрики общего плана («транспорт», «недвижимость» и пр ), а на втором - более конкретные («мотоциклы», «автомобили», «спецтехника», «квартиры», «дома», «дачные участки»).

На страницах вывода объявлений будет присутствовать пагинация. Также будет предусмотрена возможность поиска объявлений.

Под любым объявлением можно будет оставить комментарии. Эта возможность предоставляется даже не зарегистрированным пользователям.

При добавлении объявления пользователь может прикрепить любое количество изображений, одно из которых будет основным и выводиться в списке объявлений. Все изображения будут видны на странице конкретного объявления. Добавление изображений не является обязательным к размещению.

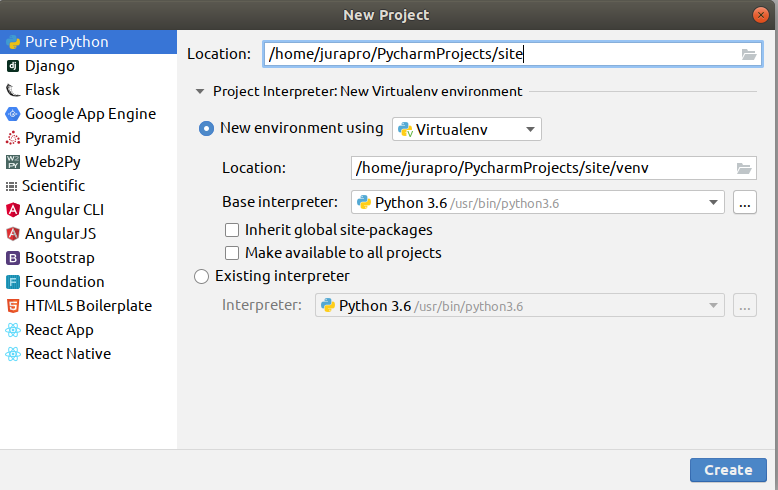
Процедура регистрации пользователя будет предусматривать активацию аккаунта через электронную почту.

Сайт будет содержать следующие страницы:

* главная - отображение последних 10 объявлений без разбиения на рубрики;
* страница списка объявлений - отображение объявлений, относящихся к данной рубрике (с использованием пагинации). Кроме того она будет содержать форму для поиска объявлений;
* страница отдельного объявления - содержит полную информацию об объявлении, а также оставленные под ним комментарии и форму для добавления нового комментария;
* страницы регистрации и активации нового пользователя;
* страницы входа и выхода;
* страница профиля пользователя - содержит список объявлений, оставленных текущим пользователем;
* страницы изменения пароля, правки и удаления пользовательского профиля;
* страницы сведения о сайте, о правах разработчика, пользовательского соглашения и пр.

## Создание нового проекта и базового приложения

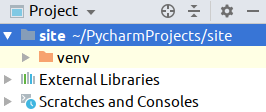
Запустим PyCharm и создадим новый пустой проект со следующими настройками (Рис 2.1):



*Рис 2.1. Базовые настройки пустого проекта в PyCharm*

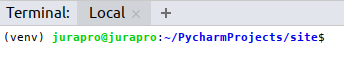
**Примечание:** Мы будем использовать виртуальную среду Virtualenv, которая позволяет изолировать проект со всеми зависимостями в одну общую директорию. Это удобно и для разработки и для дальнейшего разворачивания приложения.

Вновь созданные проект представляет из себя директорию site с директорией виртуальной среды venv внутри (Рис 2.2).



*Рис 2.2. Структура созданного проекта в PyCharm*

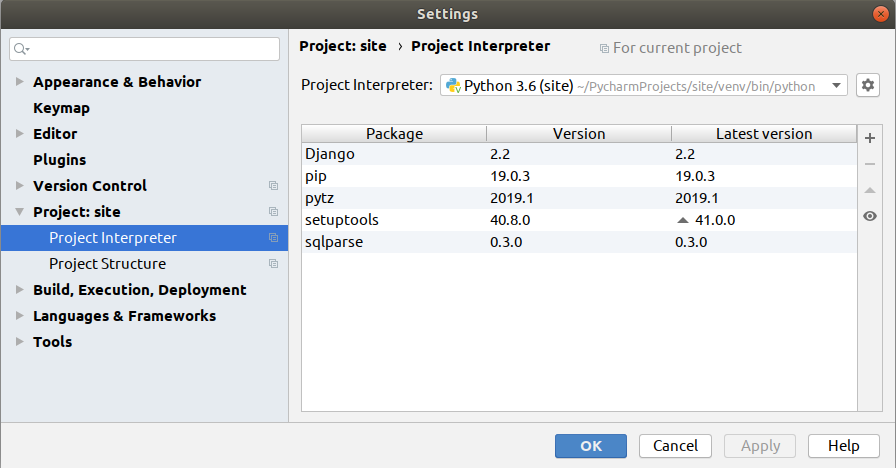
Виртуальная среда будет уже активирована, о чем свидетельствует префикс (venv) перед именем пользователя во встроенном терминале (Рис 2.3). Все манипуляции, которые мы будем проводить используя этот терминал будут изолированы внутри директории проекта.



*Рис 2.3. Терминал созданного проекта в PyCharm*

Установим фреймворк Django внутри нашего окружения. Для этого необходимо в терминале ввести команду: **pip install django**. Спустя некоторое время фреймворк и необходимые пакеты будут установлены.

**Примечание**: Альтернативным способом установки пакетов является установка из графического интерфейса PyCharm. Для этого нужно в меню File выбрать пункт Settings, затем Project:site>Project Interpreter (Рис 2.4).

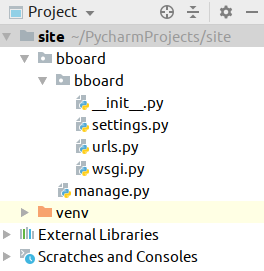


*Рис 2.4. Графический интерфейс установки пакетов в PyCharm*

После установки фреймворка можно создать проект django с помощью команды: **django-admin startproject bboard**. После того, как команда отработает, структура директорий изменится (Рис 2.5).

Откроем модуль настроек setting.py из пакета конфигурации и внесем в него следующие правки:

* изменим имя файла, в котором будет храниться база данных сайта на bboard.data;
* изменим код языка для вывода системных сообщений и страниц административного сайта на “ru-ru”.



*Рис 2.5. Структура базового проекта django*

Измененные фрагменты конфигурации должны выглядеть так:

import os

...

DATABASES = {

**'default'**: {

**'ENGINE'**: **'django.db.backends.sqlite3'**,

**'NAME'**: os.path.join(BASE\_DIR, **'bboard.data'**),

}

}

…

LANGUAGE\_CODE = **'ru-ru'**

…

## Создание основного приложения main

Создадим основное приложение main, которое реализует всю функциональность сайта. Перейдем в папку проекта и выполним команду: **python manage.py startapp main**. В проект добавился модуль main с необходимыми файлами и директориями (Рис 2.6).

Найдем модуль настроек приложения app.py и добавим в объявление конфигурационного класса MainConfig атрибут verbose\_name чтобы изменить название приложения:

class MainConfig(AppConfig):

verbose\_name = **'Электронная доска объявлений'**

name = **'main'**

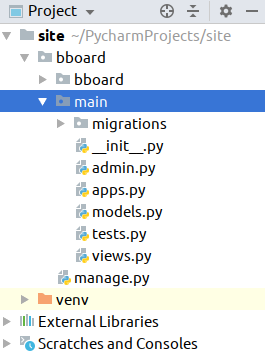
Теперь нужно зарегистрировать приложение в основном проекте. Для этого вернемся к модулю настроек settings.py и добавим его в список зарегистрированных в проекте:

INSTALLED\_APPS = [

...

**'main.apps.MainConfig'**,

]



*Рис 2.6. Структура модуля main*

## Создание базового шаблона

Создадим в пакете приложения main директорию templates, а в ней директорию layout. Там будет храниться базовый шаблон нашего приложения.

Для ускорения верстки используем CSS-фреймворк Bootstrap 4. Для того, чтобы им воспользоваться установим его командой: **pip install django-bootstrap4**.

Добавим приложение bootstrap4 в файл конфигурации settings.py:

INSTALLED\_APPS = [

...

**'bootstrap4'**,

]

Код базового шаблона представлен в листинге 2.1. Сохраним его в файле templates/layout/basic.html.

*Листинг 2.1. Базовый шаблон templates/layout/basic.html*

{% load static %}

{% load bootstrap4 %}

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<link rel="stylesheet" type="text/css" href="{% static 'main/style.css' %}">

{% bootstrap\_css %}

{% bootstrap\_javascript jquery='slim' %}

<title>{% block title %} Главная {% endblock %} - Электронная доска объявлений </title>

</head>

<body>

<div class="container-fluid">

<div class="d-flex flex-column flex-md-row p-3 px-md-4 mb-3 bg-white border-bottom box-shadow">

<h5 class="mr-md-auto p-2">Электронная доска объявлений</h5>

<div class="nav-item dropdown">

<a class="nav-link dropdown-toggle" href="#" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">Профиль</a>

<div class="dropdown-menu">

<a class="dropdown-item" href="#">Мои объявления</a>

<a class="dropdown-item" href="#">Изменить личные данные</a>

<a class="dropdown-item" href="#">Изменить пароль</a>

<div class="dropdown-divider"></div>

<a class="dropdown-item" href="#">Выйти</a>

<div class="dropdown-divider"></div>

<a class="dropdown-item" href="#">Удалить</a>

</div>

</div>

<a class="p-2 text-dark" href="#">Регистрация</a>

<a class="p-2 text-dark" href="#">Вход</a>

</div>

<div class="row">

<nav class="col-md-auto nav flex-column">

<a class="nav-link root" href="{% url 'main:index' %}">Главная</a>

<span class="nav-link root font-weight-bold">

Недвижимость

</span>

<a class="nav-link" href="#">Жилье</a>

<a class="nav-link" href="#">Скалады</a>

<a class="nav-link" href="#">Квартиры</a>

<span class="nav-link root font-weight-bold">

Транспорт

</span>

<a class="nav-link" href="#">Автомобили</a>

<a class="nav-link" href="#">Мото</a>

<a class="nav-link" href="#">Велосипеды</a>

</nav>

<div class="container">

{% bootstrap\_messages %}

{% block content %}

{% endblock %}

</div>

</div>

<footer class="mt-3 border-top">

<small class="p-2"> &copy; Методическое пособие: Знакомство с фреймворком

Django. 2018-2019

</small>

</footer>

</div>

</body>

</html>

Код базового шаблона достаточно большой, он включает в себя много стилевых классов, формирующих разметку и оформление в стиле Bootstrap. Отметим наиболее важные фрагменты кода.

* <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

Этот метатег отвечает за корректную обработку адаптивной верстки;

* <title>{% block title %} Главная {% endblock %} - Электронная доска объявлений </title>

В теге <title> создан первый блок, с названием title. Он предназначен для вывода названия у каждой страницы сайта;

* <link rel="stylesheet" type="text/css" href="{% static 'main/style.css' %}"> - Привязываем таблицу стилей static/main/style.css. Создадим ее чуть позже. В ней будут записываться некоторые специфические для нашего сайта стили;
* {% bootstrap\_css %}

{% bootstrap\_javascript jquery='slim' %}

Подключение таблицы стилей и программного кода Bootstrap;

* <div class="container-fluid">

…

</div>

Основной контейнер в котором содержится все остальное;

* <div class="d-flex flex-column flex-md-row p-3 px-md-4 mb-3 bg-white border-bottom box-shadow">

…

</div>

Контейнер для верхней части сайта. Комбинация различных классов позволяет настроить внешний вид и поведение так, как необходимо. В данном случае элементы, вложенные в этот контейнер станут flex-элементами и выстроятся в линию. При уменьшении разрешения они выстроятся вертикально. Также выбран цвет фона, отступы, рамка снизу и т.д.;

* <h5 class="mr-md-auto p-2">Электронная доска объявлений</h5>

<div class="nav-item dropdown">

…

</div>

<a class="p-2 text-dark" href="#">Регистрация</a>

...

Элементы шапки сайта. Интересным является возможность создания выпадающего меню без написания кода js;

* <div class="row">

<nav class="col-md-auto nav flex-column">

...

</nav>

<div class="container">

...

</div>

</div>

Здесь внутри контейнера с классом row содержатся два элемента. Первый отвечает за отображения вертикального меню (с соответствующим поведением элементов), а второй является контейнером для основного контента сайта;

* {% bootstrap\_messages %}

Блок кода, отвечающий за вывод системных сообщений, оформленных в Bootstrap стиле;

* {% block content %}

{% endblock %}

Блок content, в котором будет выводится основное содержимое страниц;

* <footer class="mt-3 border-top">

...

</footer>

Подвал сайта, в котором будет содержаться дополнительная информация.

Основное преимущество Bootstrap - большое количество стилей на все случаи жизни. Пользуясь ими можно решать типовые задачи по верстке исключительно средствами этого фреймворка, не написав ни единого стиля.

Однако, нам все-таки придется написать пару стилей. Создадим в папке пакета приложения директорию static, в ней - директорию main. В последнюю запишем файл style.css с кодом ниже:

.root {

font-size: larger;

}

Данный стиль - увеличивает базовый размер шрифта на сайте.

## Главная веб-страница

После того как закончили с базовым шаблоном, можно приступить к созданию шаблона главной страницы. Он будет максимально простым, чтобы проверить работоспособность сайта.

В Django реализуется модель MVT (Model-View-Template). Каждая из этих составляющих отвечает за свое. Модель (Model) - служит для манипулирования с сущностями приложения (которые обычно представлены таблицами в базе данных). Представление (View) - содержит набор функций и классов реализующих бизнес-логику приложения. Шаблон (Template) - отвечает за то, как данные будут выглядеть на странице.

**Примечание:** Перевод “Представление”, пожалуй, не самый удачный так как может ввести в заблуждение о назначении, мы чаще всего будем использовать термин “Контроллер”.

В самом простом варианте, при реализации какого-либо действия мы должны пройти следующие нехитрые шаги:

1. Сопоставить url-адрес (или шаблон адресов) с соответствующей контроллер-функцией;
2. Реализовать контроллер-функцию (или класс), в которой выполнить необходимые манипуляции с данными и связать их с соответствующим шаблоном;
3. В шаблоне отобразить полученные данные.

Для начала создадим контроллер-функцию index(), которая просто будет запускать обработку шаблона главной функции (Листинг 2.2). Нужно отметить, что контроллер-функции хранятся в файле views.py модуля приложения (в нашем случае bboard/main/views.py).

*Листинг 2.2. Код контроллер-функции index()*

from django.shortcuts import render

def index(request):

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

В директории templates создадим директорию main, в которой будем сохранять шаблоны страниц (Рис 2.7). Добавим туда шаблон главной страницы - index.html (Листинг 2.3).

**Примечание:** Для того чтобы PyCharm понимал язык шаблонизатора и работало автодополнение можно щелкнуть правой кнопкой мыши по директории с шаблонами и выбрать Mark Directory As -> Template Folder. При необходимости нужно будет выбрать язык шаблонизатора.

*Листинг 2.3. Код шаблона index.html*

{% extends "layout/basic.html" %}

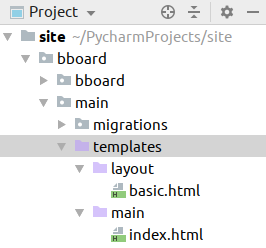
{% block content %}

<h2>Последние 10 объявлений</h2>

{% endblock %}

Теперь нужно написать маршруты. Заодно настроем приложение так, чтобы статический файлы сервера не кэшировались, так как работа над оформлением сайта еще не закончена.

Начнем со списка маршрутов уровня проекта. Откроем модуль urls.py пакета конфигурации и исправим код так, чтобы он выглядел как показано в листинге 2.4.



*Рис 2.7. Структура модуля main*

*Листинг 2.4. Код модуля urls.py пакета конфигурации (bboard/urls.py)*

from django.contrib import admin

from django.urls import path, include

from django.conf import settings

from django.views.decorators.cache import never\_cache

from django.views.static import serve

urlpatterns = [

path(**'admin/'**, admin.site.urls),

path(**''**, include(**'main.urls'**, namespace=**''**)),

]

if settings.DEBUG:

urlpatterns.append(path(**'static/<path:path>'**, never\_cache(serve)))

Здесь мы добавили к стандартному маршруту (/admin) маршрут, который ссылается на наше приложение. Так как входная строка пустая, то этот маршрут будет корневым. То есть если в списках маршрутов уровня пакета конфигурации не будет обнаружено соответствующего маршрута, то по умолчанию будет просматриваться модуль urls.py именно приложения bboard.

Также мы указали серверу не кешировать статические маршруты, если приложение запущено в режиме разработки.

Теперь нужно создать список маршрутов уровня приложения. Создадим файл urls.py в директории main и запишем в него код из листинга 2.5.

Теперь можно проверить работоспособность сайта. Запустим отладочный сервер с помощью команды: **python manage.py runserver**.

*Листинг 2.5. Код модуля urls.py пакета приложения (main/urls.py)*

from django.urls import path

from .views import index

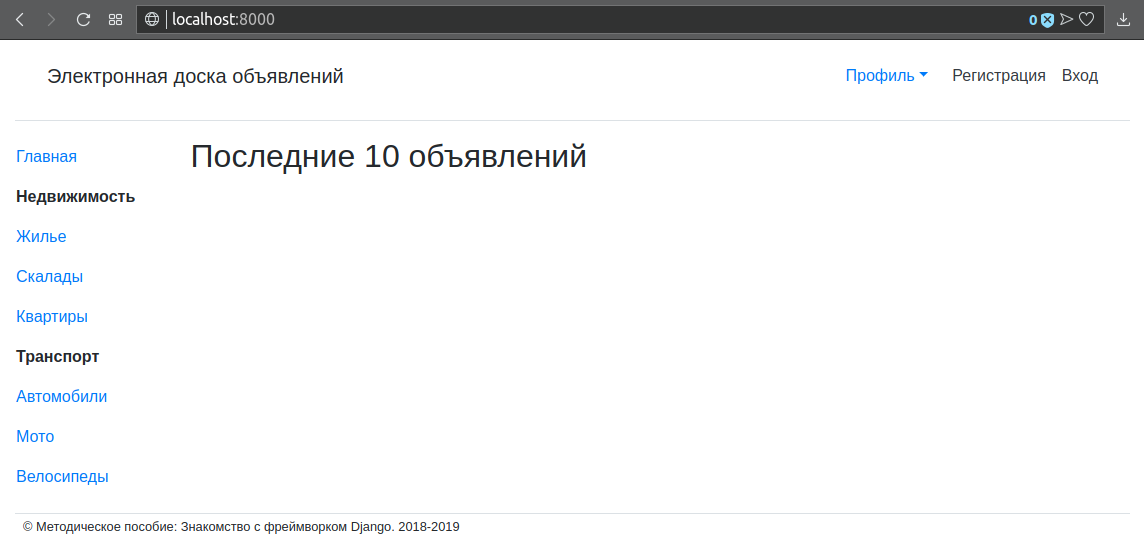
app\_name = **'main'**

urlpatterns = [

path(**''**, index, name=**'index'**)

]

Если все сделано верно, в консоли мы увидим сообщение, что веб-сервер запущен по адресу <http://127.0.0.1:8000>. Теперь если запустить браузер и перейти по соответствующему адресу, то можно попасть на главную страницу сайта (Рис 2.8).



*Рис 2.8. Главная страница сайта*

## Дополнительные веб-страницы

Сразу создадим страницы со сведениями о сайте и правах его разработчиков. Для вывода всех статичных страниц будем использовать одним контроллер и, соответственно один маршрут. Для того, чтобы контроллер мог различать страницы будем передавать какой-нибудь идентификатор страницы с URL-параметром. В качестве идентификатора используем имя шаблона без расширения.

В список маршрутов уровня приложения добавим следующий код (выделен полужирным):

from django.urls import path

from .views import index

**from .views import other\_page**

app\_name = **'main'**

urlpatterns = [

**path('<str:page>/', other\_page, name='other'),**

path(**''**, index, name=**'index'**)

]

Имя шаблона выводимой страницы мы передаем через URL-параметр page. А контроллер, выводящий вспомогательные страницы, назван other\_page и будет реализован в виде соответствующей контроллер-функции.

В модуль views.py пакета приложения добавим код контроллера-функции other\_page() (Листинг 2.6).

*Листинг 2.6. Код контроллера-функции other\_page()*

…

from django.http import HttpResponse, Http404

from django.template import TemplateDoesNotExist

from django.template.loader import get\_template

…

def other\_page(request, page):

try:

template = get\_template(**'main/'** + page + **'.html'**)

except TemplateDoesNotExist:

raise Http404

return HttpResponse(template.render(request=request))

В этой функции мы получаем имя выводимой страницы из параметра page, добавляем к нему путь и расширение, получив таким образом полный путь до шаблона и пытаемся загрузить его вызовом функции get\_template(). Если загрузка прошла успешно, мы формируем на основе этого шаблона страницу.

Если же шаблон загрузить не удалось, будет возбуждено исключение TemplateDoesNotExist. Мы перехватываем это исключение и возбуждаем другое исключение Http404, которое приведет к отправке страницы с сообщением об ошибке 404 (ресурс не найден).

Теперь необходимо создать шаблон для страницы со сведениями о самом сайте и правах его разработчиков.

Создадим шаблон templates/main/about.html с кодом из листинга 2.7.

*Листинг 2.7. Код шаблона templates/main/about.html*

{% extends "layout/basic.html" %}

{% block title %}О сайте{% endblock %}

{% block content %}

<h2>О сайте</h2>

<p>Этот сайт предназначен для публикации электронных объявлений</p>

<p>Все права оформлены и охраняются законом РФ</p>

{% endblock %}

После этого добавим в базовый шаблон templates/layout/basic.html ссылку на созданную страницу:

<a class="nav-link root" href="{% url 'main:other' page='about' %}">О сайте</a>

Если все сделано верно, то по адресу <http://localhost:8000/about/> можно увидеть созданную страницу.

# 3. Работа с пользователями

Модель - это однозначное и исчерпывающее описание сущности, хранящейся в базе данных в виде класса Python. Класс модели описывает таблицу базы данных, в которой будет храниться набор сущностей, и содержит атрибуты классы, каждый из которых описывает одно из полей этой таблицы. Можно сказать, что модель - это представление таблицы и ее полей средствами Python.

Отдельный экземпляр класса модели представляет отдельную конкретную сущность, извлеченную из базы, то есть отдельную запись соответствующий таблицы. Пользуясь объявленными в модели атрибутами класса, мы можем получать значения, хранящиеся в полях записи, равно как и записывать в них новые значения.

Помимо этого, класс модели предоставляет инструменты для выборки сущностей из базы, их фильтрации и сортировки на основе заданных критериев.

Модели объявляются на уровне приложения. Объявляющий их код должен записываться в модуль models.py пакета приложения. Изначально этот модуль пуст.

## Модель пользователя

В Django поставляется стандартная модель пользователя django.contrib.auth. Ее удобно использовать, однако она не подходит, поскольку нам нужно хранить дополнительную информацию о пользователе. Поэтому мы создадим собственную модель, сделав ее производной от стандартной абстрактной модели AbstractUser, объявленной в модуле django.contrib.auth.

Наша модель будет носить название AdvUser. Ниже представлен список полей, которые мы объявим в ней.

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Тип** | **Дополнительные параметры** | **Описание** |
| is\_activated | BooleanField | Значение по умолчанию - True, индексированное | Признак, прошел ли пользователь процедуру активации |
| send\_messages | BooleanField | Значение по умолчанию - True | Признак, желает ли пользователь получать уведомления о новых комментариях |

Создадим файл models.py в директории main. Добавим туда код модели пользователя (Листинг 3.1).

*Листинг 3.1. Код модели AdvUser*

from django.db import models

from django.contrib.auth.models import AbstractUser

class AdvUser(AbstractUser):

is\_activated = models.BooleanField(default=True, db\_index=True,

verbose\_name=**'Прошел активацию?'**)

send\_messages = models.BooleanField(default=True,

verbose\_name=**'Оповещать при новых комментариях?'**)

class Meta(AbstractUser.Meta):

pass

Укажем ее как модель пользователя, используемую подсистемой разграничения доступа Django. Для этого в файле settings.py пакета конфигурации добавим строку:

AUTH\_USER\_MODEL = **'main.AdvUser'**

Остановим отладочный веб-сервер и выполним команду на создание миграций:

**python manage.py makemigrations**

и после на их выполнение:

**python manage.py migrate**

После того как миграции будут выполнены, создадим суперпользоаателя:

**python manage.py createsuperuser**

Откроем возможность управлять нашей моделью через административный сайт. Для этого в файле admin.py пакета приложения добавим следующий код:

from .models import AdvUser

admin.site.register(AdvUser)

Вновь запустим отладочный веб-сервер (команда python manage.py runserver) и перейдем по адресу <http://localhost:8000/admin>. Введя данные только что созданного суперпользователя мы попадем в административный сайт.

## Страницы входа, профиля и выхода

### Страница входа

Для начала подготовим страницу входа. Для ее реализации используем контроллер-класс LoginView. На его основе создадим свой подкласс, в котором переопределим все необходимые для работы контоллера параметры.

Код контроллера-класса, выполняющего вход показан в листинге 3.2. Его нужно добавить в модуль views.py пакета приложения.

*Листинг 3.2. Код контроллера-класса BBLoginView*

…

from django.contrib.auth.views import LoginView

…

class BBLoginView(LoginView):

template\_name = **'main/login.html'**

Код класса очень прост. Единственное что он делает - это задает путь к файлу шаблона, занеся его в атрибут template\_name. Остальные параметры мы оставляем по умолчанию.

Запишем новый маршрут, который будет указывать на контроллер BBLoginView в списке уровня приложения. В файл urls.py пакта приложения добавим следующий код:

from .views import BBLoginView

...

urlpatterns = [

path(**'accounts/login'**, BBLoginView.as\_view(), name=**'login'**),

...

]

Здесь мы связываем с нашим новым контроллером маршрут **accounts/login/**. По умолчанию именно по этому маршруту фреймворк перенаправляет при попытке гостя получить доступ к закрытой для него странице.

Далее необходимо написать шаблон страницы входа templates/main/login.html (Листинг 3.3)

*Листинг 3.3. Код шаблона templates/main/login.html*

{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %} Вход {% endblock %}

{% block content %}

<h2>Вход</h2>

{% if user.is\_authenticated %}

<p>Вы уже выполнили вход.</p>

{% else %}

<form method="post">

{% csrf\_token %}

{% bootstrap\_form form layout='horizontal' %}

<input type="hidden" name="next" value="{{ next }}">

{% buttons submit='Войти' %} {% endbuttons %}

</form>

{% endif %}

{% endblock %}

Мы использовали для вывода формы инструменты, предлагаемые bootstrap.

Далее внесем правки в базовый шаблон. Нам необходимо, чтобы ссылки Вход и Регистрация были видны гостям, а пункт Профиль пользователям, выполнившим вход. Кроме того поправим ссылку на страницу входа на сайт.

...

<div class="d-flex flex-column flex-md-row p-3 px-md-4 mb-3 bg-white border-bottom box-shadow">

<h5 class="mr-md-auto p-2">Электронная доска объявлений</h5>

{% if user.is\_authenticated %}

<div class="nav-item dropdown">

<a class="nav-link dropdown-toggle" href="#" data-toggle="dropdown" aria-haspopup="true"

aria-expanded="false">Профиль</a>

<div class="dropdown-menu">

<a class="dropdown-item" href="#">Мои объявления</a>

<a class="dropdown-item" href="#">Изменить личные данные</a>

<a class="dropdown-item" href="#">Изменить пароль</a>

<div class="dropdown-divider"></div>

<a class="dropdown-item" href="#">Выйти</a>

<div class="dropdown-divider"></div>

<a class="dropdown-item" href="#">Удалить</a>

</div>

</div>

{% else %}

<a class="p-2 text-dark" href="#">Регистрация</a>

<a class="p-2 text-dark" href="{% url 'main:login' %}">Вход</a>

{% endif %}

<a class="nav-link root" href="{% url 'main:other' page='about' %}">О сайте</a>

</div>

…

### Страница профиля

Теперь создадим страницу пользовательского профиля. Контроллер, который будет отвечать за ее вывод реализуем в виде функции и назовем profile(). Ее код представлен в листинге 3.4.

*Листинг 3.4. Код контроллера-функции profile()*

…

from django.contrib.auth.decorators import login\_required

…

@login\_required

def profile(request):

return render(request, **'main/profile.html'**)

На страницу пользовательского профиля будет выводится список объявлений, оставленных пользователем. Но так как объявлений пока нет, выводить нечего.

Поскольку страница пользовательского профиля должна быть доступна только зарегистрированным пользователям, успешно выполнившим вход на сайт, функция profile() помечена декоратором login\_required().

Добавим в список маршрутов уровня приложения маршрут, который укажет на функцию profile():

from .views import profile

...

urlpatterns = [

path(**'accounts/profile/'**, profile, name=**'profile'**),

...

]

Код шаблона templates/main/profile.html представлен в листинге 3.5

*Листинг 3.5. Код шаблона templates/main/profile.html*

{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %} Профиль пользователя {% endblock %}

{% block content %}

<h2>Профиль пользователя {{ user.username }}</h2>

{% if user.first\_name and user.last\_name %}

<p>Здравствуйте, {{ user.first\_name }} {{ user.last\_name }} </p>

{% else %}

<p>Здравствуйте! </p>

{% endif %}

<h3>Ваши объявления</h3>

{% endblock %}

Если пользователь при регистрации написал свои имя и фамилию, то на странице будет выведено персонализированное приветствие, если же пользователь не ввел эти данные, то простое приветствие.

Осталось в базовом шаблоне найти гиперссылку **Мои объявления** и вставить адрес страницы профиля:

<a class="dropdown-item" href="{% url 'main:profile' %}">Мои объявления</a>

### Страница выхода

Для создания страницы выходи мы используем класс BBLogoutView, производный от класса LogoutView. Код класса показан в листинге 3.6.

*Листинг 3.6. Код контроллер-класса BBLogoutView*

from django.contrib.auth.views import LogoutView

from django.contrib.auth.mixins import LoginRequiredMixin

…

class BBLogoutView(LoginRequiredMixin, LogoutView):

template\_name = **'main/logout.html'**

…

Для того, чтобы страница выхода была доступна только зарегистрированным пользователям, мы добавили в число суперклассов класса BBLogoutView примесь LoginRequiredMixin.

Запишем маршрут, ведущий на этот контроллер:

from .views import BBLogoutView

...

urlpatterns = [

path(**'accounts/logout/'**, BBLogoutView.as\_view(), name=**'logout'**),

...

]

Напишем шаблон templates/main/logout.html (Листинг 3.7)

*Листинг 3.7. Код шаблона templates/main/logout.html*

{% extends "layout/basic.html" %}

{% block title %} Выход {% endblock %}

{% block content %}

<h2>Выход</h2>

<p>Вы успешно вышли с сайта.</p>

{% endblock %}

В базовом шаблоне изменим ссылку **Выход**:

<a class="dropdown-item" href="{% url 'main:logout' %}">Выйти</a>

## Страницы редактирования личных данных пользователя

### Страница правки основных сведений

Страница позволит пользователю исправить его имя (логин), адрес электронной почты, реальное имя, фамилию и возможность включить или отключить оповещения о новых комментариях.

Для ввода данных язык HTML предлагает использовать веб-формы. Их создание и обработка на стороне сервера довольно сложный и трудоемкий процесс. Благо, что в Django присутствуют специальные классы - **формы, связанные с моделью**. Такие формы умеют генерировать теги, что создадут входящие в состав формы элементы управления, проверять на корректность введенные данные и сохранять их в модели с которой она связана.

Для решения нашей задачи объявим форму, связанную с моделью AdvUser. В эту форму, которую мы назовем ChangeUserInfoForm, пользователь будет вносить новые личные данные. Формы обычно помещаются в файле forms.py пакета приложения. Создадим этот файл и запишем туда код формы ChangeUserInfoForm (Листинг 3.8).

*Листинг 3.8. Код формы ChangeUserInfoForm*

from django import forms

from .models import AdvUser

class ChangeUserInfoForm(forms.ModelForm):

email = forms.EmailField(required=True,

label=**'Адрес электронной почты'**)

class Meta:

model = AdvUser

fields = (**'username'**, **'email'**, **'first\_name'**, **'last\_name'**,

**'send\_messages'**)

Мы скомбинировали быстрое и полное объявление формы. Так как необходимо, чтобы пользователь обязательно вносил значение в поле email модели AdvUser, то выполняем полное объявление поля email формы. Так как параметры остальных полей не меняются, в их отношении мы применим быстрое объявление.

Теперь необходимо создать контроллер, который будет выполнять правку записи модели. Его мы напишем на базе высокоуровневого класса UpdateView. Готовый код контроллера-класса ChangeUserInfoView представлен в листинге 3.9.

*Листинг 3.9. Код контроллера-класса ChangeUserInfoView*

…

from django.contrib.messages.views import SuccessMessageMixin

from django.shortcuts import get\_object\_or\_404

from django.urls import reverse\_lazy

from django.views.generic import UpdateView

from .forms import ChangeUserInfoForm

from .models import AdvUser

…

class ChangeUserInfoView(SuccessMessageMixin, LoginRequiredMixin,

UpdateView):

model = AdvUser

template\_name = **'main/change\_user\_info.html'**

form\_class = ChangeUserInfoForm

success\_url = reverse\_lazy(**'main:profile'**)

success\_message = **'Личные данные пользователя изменены'**

def dispatch(self, request, \*args, \*\*kwargs):

self.user\_id = request.user.pk

return super().dispatch(request, \*args, \*\*kwargs)

def get\_object(self, queryset=None):

if not queryset:

queryset = self.get\_queryset()

return get\_object\_or\_404(queryset, pk=self.user\_id)

В процессе работы этот контроллер должен извлечь из модели AdvUser запись, представляющую текущего пользователя, для чего ему необходимо предварительно получить идентификатор текущего пользователя. А получить его можно из объекта текущего пользователя, хранящегося в атрибуте user объекта запроса.

Пожалуй, наилучшее место для получения ключа текущего пользователя - метод dispatch(), наследуемый всеми контроллерами-классами от их общего суперкласса View. Это метод выполняется в самом начале исполнения контроллера-класса и получает объект запроса в качестве одного из параметров. В предопределенном методе мы извлекаем ключ пользователя и сохраним его в атрибуте user\_id.

Извлечение исправляемой записи выполняется в методе get\_obect(), которую контроллер-класс унаследовал от примеси SingleObjectMixin. В переопределенном методе учитываем тот момент, что набор записей, из которого нужно извлечь запись может быть передан методу с параметром queryset, а может быть и не передан - в этом случае набор записей получаем вызовом метода get\_queryset(). После чего непосредственно ищим запись, представляющую текущего пользователя.

В качестве одного из суперклассов этого контроллера мы указали примесь LoginReauiredMixin, запрещающую доступ к контроллеру гостям и примесь SuccessMessageMixin, которая применяется для вывода всплывающих сообщений об успешном выполнении операции.

Теперь необходимо добавить соответствующий маршрут в модуль urls.py пакета приложения:

from .views import ChangeUserInfoView

...

urlpatterns = [

...

path(**'accounts/profile/change/'**, ChangeUserInfoView.as\_view(), name=**'profile\_change'**),

path(**'accounts/profile/'**, profile, name=**'profile'**),

...

]

Код шаблона templates/main/change\_user\_info.html, создающего страницу, показан в листинге 3.10.

*Листинг 3.10. Код шаблона templates/main/change\_user\_info.html*

{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %} Правка личных данных {% endblock %}

{% block content %}

<h2>Правка личных данных пользователя {{ user.username }}</h2>

<form method="post">

{% csrf\_token %}

{% bootstrap\_form form layout='horizontal' %}

{% buttons submit='Сохранить' %} {% endbuttons %}

</form>

{% endblock %}

В базовом шаблоне изменим ссылку **Изменить личные данные**:

<a class="dropdown-item" href="{% url 'main:profile\_change' %}">Изменить личные данные</a>

### Страница изменения пароля

Реализовать функцию смены пароля довольно просто. Контроллер мы делаем на основе контроллера-класса PasswordChangeView, который реализует смену пароля. Код нашего контроллера-класса показан в листинге 3.11.

*Листинг 3.11. Код контроллера-класса BBPasswordChangeView*

from django.contrib.auth.views import PasswordChangeView

…

class BBPasswordChangeView(SuccessMessageMixin, LoginRequiredMixin,

PasswordChangeView):

template\_name = **'main/password\_change.html'**

success\_url = reverse\_lazy(**'main:profile'**)

success\_message = **'Пароль пользователя изменен'**

Добавим новый маршрут в файл urls.py:

from .views import BBPasswordChangeView

...

urlpatterns = [

...

path(**'accounts/password/change/'**, BBPasswordChangeView.as\_view(), name=**'password\_change'**),

...

]

Ниже представлен код шаблона страницы для смены пароля templates/main/password\_change.html (Листинг 3.12).

*Листинг 3.12. Код шаблона templates/main/password\_change.html*

{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %} Смена пароля {% endblock %}

{% block content %}

<h2>Смена пароля пользователя {{ user.username }}</h2>

<form method="post">

{% csrf\_token %}

{% bootstrap\_form form layout='horizontal' %}

{% buttons submit='Сменить пароль' %} {% endbuttons %}

</form>

{% endblock %}

В базовом шаблоне изменим ссылку **Изменить пароль**:

<a class="dropdown-item" href="{% url 'main:password\_change' %}">Изменить пароль</a>

## Страницы регистрации и активации пользователей

Для выполнения регистрации нам нужно написать форму для ввода сведений о новом пользователе, контроллеры и шаблоны для страниц непосредственно регистрации и подтверждения об успешной регистрации.

Для отправки письма о необходимости регистрации мы объявим свой **сигнал**.

Сигнал - это сущность, создаваемая Django при выполнении какого-либо действия: создании новой записи в модели, удалении записи, входе пользователя на сайт, выходе и прочее. К сигналу можно привязать **обработчик** - функцию или метод, который будет вызываться при возникновении сигнала.

Сигналы предоставляют возможность вклиниться в процесс работы самого фреймворка или отдельный приложений - неважно, стандартных или написанных кем-то - и произвести дополнительные действия.

Для наших целей создадим сигнал user\_registrated и он получит в качестве параметра instance объект вновь созданного пользователя.

Сигнал будет объявлен в модуле models.py пакета приложения. Этот модуль выполняется непосредственно при инициализации приложения и является идеальным местом для записи кода, объявляющего сигналы.

### Форма для занесения сведений о новом пользователе

Класс формы назовем RegisterUserForm. Его код показан в листинге 3.13.

*Листинг 3.13. Код формы RegisterUserForm*

…

from django.contrib.auth import password\_validation

from django.core.exceptions import ValidationError

from .models import user\_registrated

class RegisterUserForm(forms.ModelForm):

email = forms.EmailField(required=True,

label=**'Адрес электронной почты'**)

password1 = forms.CharField(label=**'Пароль'**,

widget=forms.PasswordInput,

help\_text=password\_validation.password\_validators\_help\_text\_html())

password2 = forms.CharField(label=**'Пароль (повторно)'**,

widget=forms.PasswordInput,

help\_text=**'Повторите тот же самый пароль еще раз'**)

def clean\_password1(self):

password1 = self.cleaned\_data[**'password1'**]

if password1:

password\_validation.validate\_password(password1)

return password1

def clean(self):

super().clean()

password1 = self.cleaned\_data[**'password1'**]

password2 = self.cleaned\_data[**'password2'**]

if password1 and password2 and password1 != password2:

errors = {**'password2'**: ValidationError(

**'Введенные пароли не совпадают'**, code=**'password\_mismatch'**

)}

raise ValidationError(errors)

def save(self, commit=True):

user = super().save(commit=False)

user.set\_password(self.cleaned\_data[**'password1'**])

user.is\_active = False

user.is\_activated = False

if commit:

user.save()

user\_registrated.send(RegisterUserForm, instance=user)

return user

class Meta:

model = AdvUser

fields = (**'username'**, **'email'**, **'password1'**, **'password2'**,

**'first\_name'**, **'last\_name'**, **'send\_messages'**)

Здесь мы опять комбинируем быстрое и полное объявление полей. Полное объявление мы используем для создания электронной почты и обоих полей для занесения пароля. Согласно общепринятой практике, мы отведем для занесения пароля два поля.

В качестве дополнительного поясняющего текста у первого поля пароля мы используем объединенный текст с требованиями к вводимому паролю, предоставленный всеми доступными в системе валидаторами.

В методе clean\_password1() мы выполняем валидацию поля, введенного в первое поле, с применением доступных в системе валидаторов пароля. Проверять таким образом второй пароль нет смысла, так как они должны будут быть одинаковыми.

В переопределенном методе clean() мы проверяем, совпадают ли оба введенных пароля. Эта проверка будет выполняться после проверки на корректность пароля из первого поля.

При сохранении нового пользователя мы заносим значения False в поля is\_active (является ли пользователь активным) и is\_activated (выполнил ли пользователь процедуру активации), там самым мы сообщаем фреймворку, что этот пользователь еще не может выполнять вход на сайт. Также нам нужно записать в модель закодированный пароль и отправить сигнал user\_registrated, чтобы отослать пользователю письмо с требованием активации. Все это мы выполняем в переопределенном методе save().

### Контроллеры и шаблоны для регистрации пользователя

Контроллер-класс, регистрирующий пользователя, мы назовем RegisterUserView и сделаем производным от контроллера-класса CreateView. Код нашего контроллера показан в листинге 3.14.

*Листинг 3.14. Код контроллер-класса RegisterUserView*

…

from django.views.generic import CreateView

from .forms import RegisterUserForm

…

class RegisterUserView(CreateView):

model = AdvUser

template\_name = **'main/register\_user.html'**

form\_class = RegisterUserForm

success\_url = reverse\_lazy(**'main:register\_done'**)

Контроллер, который выведет сообщение об успешной регистрации, будет называться RegisterDoneView и так как будет очень прост, то станет производным от класса TemplateView (Листинг 3.15).

*Листинг 3.15. Код контроллер-класса RegisterDoneView*

from django.views.generic.base import TemplateView

class RegisterDoneView(TemplateView):

template\_name = **'main/register\_done.html'**

Откроем модуль urls.py пакета приложеня, где записываются маршруты и вставим в этот список два маршрута, ведущих на только что созданные контроллеры:

from .views import RegisterUserView, RegisterDoneView

...

urlpatterns = [

path(**'accounts/register/done/'**, RegisterDoneView.as\_view(), name=**'register\_done'**),

path(**'accounts/register/'**, RegisterUserView.as\_view(), name=**'register'**),

path(**'accounts/logout/'**, BBLogoutView.as\_view(), name=**'logout'**),

...

]

Код шаблонов страницы регистрации и сообщения о ее успешном прохождении представлен в листингах 3.16 и 3.17.

*Листинг 3.16. Код шаблона templates/main/register\_user.html*

{% extends "layout/basic.html" %}

{% load bootstrap4 %}

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

{% block content %}

<h2>Регистрация нового пользователя</h2>

<form method="post">

{% csrf\_token %}

{% bootstrap\_form form layout='horizontal' %}

{% buttons submit='Зарегистрироваться' %} {% endbuttons %}

</form>

{% endblock %}

*Листинг 3.17. Код шаблона templates/main/register\_done.html*

{% extends "layout/basic.html" %}

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

{% block content %}

<h2>Регистрация</h2>

<p>Регистрация пользователя завершена</p>

<p>На адрес электронной почты, указанный пользователем, выслано письмо для активации</p>

{% endblock %}

В базовом шаблоне изменим ссылку **Регистрация**:

<a class="p-2 text-dark" href="{% url 'main:register' %}">Регистрация</a>

### Средства для отправки писем с требованием активации

Чтобы регистрация заработала, нам необходимо написать код, который выполнит отправку электронных писем с оповещением о необходимости выполнить активацию.

Непосредственно рассылку писем будет выполнять функция send\_activation\_notification(), которую объявим позже, во вновь созданном модуле utilities.py.

Откроем модуль models.py пакета приложения и запишем в него код, который объявит сигнал user\_registrated и привяжет к нему обработчик:

from django.dispatch import Signal

from .utilities import send\_activation\_notification

user\_registrated = Signal(providing\_args=[**'instance'**])

def user\_registrated\_dispatcher(sender, \*\*kwargs):

send\_activation\_notification(kwargs[**'instance'**])

user\_registrated.connect(user\_registrated\_dispatcher)

class AdvUser(AbstractUser):

is\_activated = models.BooleanField(default=True, db\_index=True,

verbose\_name=**'Прошел активацию?'**)

send\_messages = models.BooleanField(default=True,

verbose\_name=**'Оповещать при новых комментариях?'**)

class Meta(AbstractUser.Meta):

pass

Создадим в пакете приложения модуль utilities.py. Запишем в него код из листинга 3.18.

*Листинг 3.18. Код реализующий отправку писем с оповещениями об активации*

from django.template.loader import render\_to\_string

from django.core.signing import Signer

from bboard.settings import ALLOWED\_HOSTS

signer = Signer()

def send\_activation\_notification(user):

if ALLOWED\_HOSTS:

host = **'http://'** + ALLOWED\_HOSTS[0]

else:

host = **'http://localhost:8000'**

context = {**'user'**: user, **'host'**: host, **'sign'**: signer.sign(user.username)}

subject = render\_to\_string(**'email/activation\_letter\_subject.txt'**,

context)

body\_text = render\_to\_string(**'email/activation\_letter\_body.txt'**,

context)

user.email\_user(subject, body\_text)

Чтобы сформировать интернет-адрес, ведущий на страницу подтверждения активации, нам понадобиться домен на котором находится ваш сайт, а во-вторых, некоторое значение уникально идентифицирующее только что зарегистрированного пользователя и при этом устойчивое к попыткам его подделать.

Домен мы извлекаем из списка разрешенных доменов, который записан в параметра ALLOWED\_HOSTS настроек проекта. Если же список пуст, мы используем локальный адрес отладочного веб-сервера.

В качестве уникального и стойкого к подделка идентификатора пользователя используем его имя, защищенное цифровой подписью. Цифровой подпись сделаем с помощью класса Signer.

Текст тела и темы письма формируем с применением шаблонов templates/email/activation\_letter\_body.txt и templates/email/activation\_letter\_subject.txt соответственно. Коды этих шаблонов представлены в листингах 3.19 и 3.20.

*Листинг 3.19. Код шаблона templates/email/activation\_letter\_body.txt*

Уважаемый пользователь {{ user.username }}

Вы зарегистрировались на сайте "Электронная доска объявлений".

Вам необходимо выполнить активацию, чтобы подтвердить свою личность.

Для этого пройдите, пожалуйста, по ссылке

{{ host }}{% url 'main:register\_activate' sign=sign %}

До свидания!

С уважением, администрация сайта.

*Листинг 3.20. Код шаблона templates/email/activation\_letter\_subject.txt*

Активация пользователя {{ user.username }}

**Примечание**: Для того, чтобы отправка писем работала, нужно в настройках проекта в файле settings.py указать параметры почтового сервера. Например:

…

EMAIL\_USE\_TLS = True

EMAIL\_HOST = **'smtp.gmail.com'**

EMAIL\_HOST\_USER = ‘**username@gmail.com'**

EMAIL\_HOST\_PASSWORD = ‘**password’**

EMAIL\_PORT = 587

### Страницы активации пользователя

Чтобы реализовать активацию нового пользователя, напишем один контроллер и три шаблона. Они создадут страницы с сообщением об успешной активации, о том, что активация была выполнена ранее и о том, что цифровая подпись у идентификатора пользователя, полученного в составе интернет-адреса, скомпрометирована.

Контроллер будет реализован в виде функции user\_activate() (Листинг 3.21)

*Листинг 3.21. Код контроллера-функции user\_activate()*

*…*

from django.core.signing import BadSignature

from .utilities import signer

…

def user\_activate(request, sign):

try:

username = signer.unsign(sign)

except BadSignature:

return render(request, **'main/bad\_signature.html'**)

user = get\_object\_or\_404(AdvUser, username=username)

if user.is\_activated:

template = **'main/user\_is\_activated.html'**

else:

template = **'main/activation\_done.html'**

user.is\_activated = True

user.is\_active = True

user.save()

return render(request, template)

Подписанный идентификатор пользователя получаем в составе интернет-адреса с параметром sign. Далее извлекаем из него имя пользователя, ищем пользователя с этим именем, делаем его активным и выводим сообщение об успешной активации. Если цифровая подпись оказалась скомпрометирована, то выводим сообщение о невозможности операции, а если пользователь уже активирован - страницу с сообщением, что активация уже произошла.

Для обработки подписанного значения используем экземпляр класса Signer, созданный в модуле utilities.py и хранящийся в переменной signer.

В списке маршрутов уровня приложения запишем маршрут, ведущий на этот контроллер:

from .views import user\_activate

...

urlpatterns = [

path(**'accounts/register/activate/<str:sign>/'**, user\_activate, name=**'register\_activate'**),

path(**'accounts/register/done/'**, RegisterDoneView.as\_view(), name=**'register\_done'**),

...

]

Код шаблонов template/main/activation\_done.html, template/main/bad\_signature.html, template/main/user\_is\_activated.html показан в листингах 3.22-3.24.

*Листинг 3.22. Код шаблона template/main/activation\_done.html*

{% extends "layout/basic.html" %}

{% block title %} Активация выполнена {% endblock %}

{% block content %}

<h2>Активация</h2>

<p>Активация пользователя завершена</p>

<p><a href="{% url 'main:login' %}">Войти на сайт</a></p>

{% endblock %}

*Листинг 3.23. Код шаблона template/main/bad\_signature.html*

{% extends "layout/basic.html" %}

{% block title %} Ошибка при активации {% endblock %}

{% block content %}

<h2>Активация</h2>

<p>Регистрация пользователя прошла неудачно.</p>

<p><a href="{% url 'main:register' %}">Зарегистрироваться повторно</a></p>

{% endblock %}

*Листинг 3.24. Код шаблона template/main/user\_is\_activated.html*

{% extends "layout/basic.html" %}

{% block title %} Пользователь уже активирован {% endblock %}

{% block content %}

<h2>Активация</h2>

<p>Пользователь с таким именем уже был активирован ранее.</p>

<p><a href="{% url 'main:login' %}">Войти на сайт</a></p>

{% endblock %}

Сохраним файлы с исходным кодом, перейдем на страницу регистрации, и пройдем всю процедуру регистрации, чтобы проверить работоспособность нашего кода.

Добавим таким образом несколько пользователей - они пригодятся для реализации удаления пользователей.

## Страница удаления пользователя

Контроллер-класс DeleteUserView, который будет выполнять удаление текущего пользователя, создадим как производный от класса DeleteView. Его код можно увидеть в листинге 3.25.

*Листинг 3.25. Код контроллер-класса DeleteUserView*

…

from django.views.generic import UpdateView, CreateView, DeleteView

from django.contrib.auth import logout

from django.contrib import messages

…

class DeleteUserView(LoginRequiredMixin, DeleteView):

model = AdvUser

template\_name = **'main/delete\_user.html'**

success\_url = reverse\_lazy(**'main:index'**)

def dispatch(self, request, \*args, \*\*kwargs):

self.user\_id = request.user.pk

return super().dispatch(request, \*args, \*\*kwargs)

def post(self, request, \*args, \*\*kwargs):

logout(request)

messages.add\_message(request, messages.SUCCESS, **'Пользователь удален'**)

return super().post(request, \*args, \*\*kwargs)

def get\_object(self, queryset=None):

if not queryset:

queryset = self.get\_queryset()

return get\_object\_or\_404(queryset, pk=self.user\_id)

В данном контроллере были использованы те же приемы, что и в контроллере ChangeUserInfoView. В методе dispatch() сохранили ключ текущего пользователя, а в методе get\_obect() отыскали по этому ключу пользователя, подлежащего удалению.

Кроме того, во-первых, перед удалением пользователя необходимо выполнить выход, что и было сделано в методе post(), а во-вторых, так как после выхода всплывающее сообщение, созданное с помощью SuccessMesageMixin пропадет, нам пришлось создать всплывающее сообщение об успешном удалении пользователя самостоятельно в том же методе.

Осталось добавить контроллер в список маршрутов уровня приложения:

**from .views import DeleteUserView**

...

urlpatterns = [

..

**path('accounts/profile/delete/', DeleteUserView.as\_view(), name='profile\_delete'),**

path(**'accounts/profile/change/'**, ChangeUserInfoView.as\_view(), name=**'profile\_change'**),

...

]

Напишем шаблон templates/main/delete\_user.html для удаления пользователя (Листинг 3.26)

*Листинг 3.26. Код шаблона templates/main/delete\_user.html*

{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %} Удаление пользователя {% endblock %}

{% block content %}

<h2>Удаление пользователя {{ object.username }}</h2>

<form method="post">

{% csrf\_token %}

{% buttons submit='Удалить' %} {% endbuttons %}

</form>

{% endblock %}

В базовом шаблоне изменим ссылку **Удалить**:

<a class="dropdown-item" href="{% url 'main:profile\_delete' %}">Удалить</a>

Теперь можно войти на сайт от имени одного из зарегистрированных пользователей и выполнить его удаление.

## Инструменты для администрирования пользователей

Осталось только подготовить инструменты для администрирования пользователей, а именно редактор, посредством которого администрация сайта будет работать с зарегистрированными пользователями. В редактор добавим возможность фильтрации пользователей по именам, адресам электронной почты, настоящим именам и фамилиям. Также реализуем вывод пользователей, уже выполнивших активацию, не выполнивших ее в течение 3 дней и недели. Также добавим в редактор действие, выполняющее отправку выбранным в списке пользователям электронных писем с уведомлением о необходимости выполнить активацию.

Класс редактора назовем AdvUserAdmin. Его код (Листинг 3.27) поместим в модуль admin.py пакта приложения, заменив им имеющийся там код.

*Листинг 3.27. Код редактора AdvUserAdmin*

from django.contrib import admin

import datetime

from .models import AdvUser

from .utilities import send\_activation\_notification

def send\_activation\_notifications(modeladmin, request, queryset):

for rec in queryset:

if not rec.is\_activated:

send\_activation\_notification(rec)

modeladmin.message\_user(request, **'Письма с оповещениями отправлены'**)

send\_activation\_notifications.short\_description = **'Отправка писем с оповещениями об активации'**

class NoactivatedFilter(admin.SimpleListFilter):

title = **'Прошли активацию?'**

parameter\_name = **'actstate'**

def lookups(self, request, model\_admin):

return (

(**'activated'**, **'Прошли'**),

(**'threedays'**, **'Не прошли более 3 дней'**),

(**'week'**, **'Не прошли более недели'**),

)

def queryset(self, request, queryset):

val = self.value()

if val == **'activated'**:

return queryset.filter(is\_active=True, is\_activated=True)

elif val == **'threedays'**:

d = datetime.date.today() - datetime.timedelta(days=3)

return queryset.filter(is\_active=False, is\_activated=False,

date\_joined\_\_date\_\_lt=d)

elif val == **'week'**:

d = datetime.date.today() - datetime.timedelta(weeks=1)

return queryset.filter(is\_active=False, is\_activated=False,

date\_joined\_\_date\_\_lt=d)

class AdvUserAdmin(admin.ModelAdmin):

list\_display = (**'\_\_str\_\_'**, **'is\_activated'**, **'date\_joined'**)

search\_fields = (**'username'**, **'email'**, **'first\_name'**, **'last\_name'**)

list\_filter = (NoactivatedFilter,)

fields = (

(**'username'**, **'email'**),

(**'first\_name'**, **'last\_name'**),

(**'is\_staff'**, **'is\_superuser'**),

(**'last\_login'**, **'date\_joined'**)

)

readonly\_fields = (**'last\_login'**, **'date\_joined'**,)

actions = (send\_activation\_notifications,)

admin.site.register(AdvUser, AdvUserAdmin)

В списке записей указываем выводить строковое представление записи, поле признака, выполнил ли пользователь активацию, дату и время его регистрации. Также разрешаем выполнять фильтрацию по полям имени, адреса электронной почты, настоящих имени и фамилии.

Для выполнения фильтрации пользователей, выполнивших активацию, не выполнивших в течении 3 дней и недели, был использован класс NoactivatedFilter.

Были явно указаны поля, которые должны выводиться в формах для правки пользователей, чтобы выстроить их в удобном для работы порядке. Даты регистрации и последнего входа сделаны доступными только для чтения.

Наконец, было зарегистрировано действие, которое рассылает пользователям письма с предписаниями выполнить активацию. Это действие реализовано функцией send\_activation\_notifications(). В ней перебираются выбранные пользователи и для каждого, кто не выполнил активацию, вызывается функция send\_activation\_notification(), описанная ранее в модуле utilities.py.

**Примечание**: В качестве самостоятельного задания реализуйте функцию сброса пароля и авторизацию через социальную сеть “ВКонтакте”.

# 4. Рубрики

Приступим к разработке функционала рубрик. Для удобства реализуем двухуровневую структуру рубрик: более общие рубрики верхнего уровня и вложенные в них рубрики. Для простоты назовем их надрубриками и подрубриками.

Для начала напишем базовую модель, в которой будут хранится оба вида рубрик, две производные **прокси-модели**: для надрубрик и продрубрик, а также редакторы для административного сайта. Кроме того напишем код, который будет выводить список рубрик в вертикальной панели навигации на каждой странице сайта.

**Примечание**: Прокси-модель - это модель, которая позволяет менять функциональность модели, но не набор объявленных в ней полей.

## Модели рубрик

### Базовая модель рубрик

Базовой модели, в которой будут храниться и надрубрики и подрубрики, дадим имя Rubric. Ее структура показана в таблице ниже:

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Тип** | **Дополнительные параметры** | **Описание** |
| name | CharField | Длина - 20 символов, индексированное | Название |
| order | IntegerField | Значение по умолчанию - 0, индексированное | Порядок |
| super\_rubric | ForeignKey | Необязательное, запрет каскадного удаления | Надрубрика |

В первом поле будет хранится название рубрики. Во-втором, целое число, обозначающее порядок следования рубрик друг за другом: при выводе рубрики сначала будут сортироваться по возрастанию значения порядка, а уже потом - по названиям.

Третье поле с именем super\_rubric будет хранить надрубрику, к которой относится текущая подрубрика. Оно будет иметь следующие важные особенности:

* это поле заполняется только в том случае, если запись хранит подрубрику;
* нужно обязательно запретить каскадное удаление, чтобы пользователь по ошибке не удалил надрубрику вместе со всеми подрубриками;
* связь, создаваемая этим полем, должна устанавливаться с моделью надрубрик, которую объявим позже. Условимся, что назовем эту модель SuperRubric,

Код класса Rubric показан в листинге 4.1. Напомним, что код все моделей заносится в модуль models.py пакета приложения.

*Листинг 4.1. Код модели Rubric*

class Rubric(models.Model):

name = models.CharField(max\_length=20, db\_index=True, unique=True,

verbose\_name=**'Название'**)

order = models.SmallIntegerField(default=0, db\_index=True,

verbose\_name=**'Порядок'**)

super\_rubric = models.ForeignKey(**'SuperRubric'**,

on\_delete=models.PROTECT, null=True, blank=True,

verbose\_name=**'Надрубрика'**)

Никаких параметрова самой модели не задаем, так как пользователи не будут работать непосредственно с этой моделью.

### Модель надрубрик

Для работы с надрубриками объявим модель SuperRubric - прокси-модель производную от Rubric. Сделаем так, чтобы она обрабатывала только надрубрики.

Чтобы изменить состав обрабатываемых моделью записей, нужно задать для нее свой диспетчер записей, который у укажет необходимые условия фильтрации. Следовательно, нужно еще и написать класс диспетчера записей.

**Примечание:** Диспетчер записей - это объект, предоставляющий доступ к набору записей, которые хранятся в модели. По умолчанию он представляет собой экземпляр класса Manager из модуля django.db.models и хранится в атрибуте objects модели. Создание своих диспетчеров записей выполняется путем наследования от класса Manager. В получившимся подклассе можно как переопределить имеющиеся методы, так и объявить новые.

Код модели SuperRubric и диспетчер записей SuperRubricManager - представлен в листинге 4.2.

*Листинг 4.2. Код модели SuperRubric и диспетчера записей SuperRubricManager*

class SuperRubricManager(models.Manager):

def get\_queryset(self):

return super().get\_queryset().filter(super\_rubric\_\_isnull=True)

class SuperRubric(Rubric):

object = SuperRubricManager()

def \_\_str\_\_(self):

return self.name

class Meta:

proxy = True

ordering = (**'order'**, **'name'**)

verbose\_name = **'Надрубрика'**

verbose\_name\_plural = **'Надрубрики'**

Условия фильтрации записей указываем в переопределенном методе qet\_queryset() класса диспетчера записей SuperRubricManager. Выбираем только записи с пустым полем super\_rubric, то есть надрубрики.

В самом класса модели SuperRubric задаем диспетчер записей SuperRubricManager в качестве основного. И не забываем объявить метод \_\_str\_\_(), который станет генерировать строковое представление надрубрики - ее название.

Также устанавливаем для модели порядок сортировки записей сначала по возрастанию значения порядка, а потом по названию.

### Модель подрубрик

Модель подрубрик SubRubric создадим тем же образом, только диспетчер записей, который объявим для нее, будет возвращать лишь подрубрики. Код соответствующих классов можно увидеть в листинге 4.3.

*Листинг 4.3. Код модели SubRubric и диспетчера записей SubRubricManager*

class SubRubricManager(models.Manager):

def get\_queryset(self):

return super().get\_queryset().filter(super\_rubric\_\_isnull=False)

class SubRubric(Rubric):

object = SubRubricManager()

def \_\_str\_\_(self):

return **'%s - %s'** % (self.super\_rubric, self.name)

class Meta:

proxy = True

ordering = (**'super\_rubric\_\_order'**, **'super\_rubric\_\_name'**, **'order'**, **'name'**)

verbose\_name = **'Подрубрика'**

verbose\_name\_plural = **'Подрубрики'**

Диспетчер записей SubRubricManager будет отбирать лишь записи с непустым полем super\_rubric. Строковое представление будет выполнено в формате <название надрубрики> - <название подрубрики>. А сортировку записей укажем по порядку надрубрики, названию надрубрики, порядку подрубрики и названию подрубрики.

Описав все необходимые классы произвести создание и выполнение миграций:

**python manage.py makemigrations**

**python manage.py migrate**

## Инструменты для администрирования рубрик

Так как вся работа над надрубриками и подрубриками будет проводиться средствами административного сайта, понадобится написать код для обеих наших моделей редакторы и некоторые вспомогательные классы.

Для надрубрик создадим встроенный редактор, чтобы пользователь, добавив новую надрубрику, смог сразу же заполнить ее подрубриками. Из формы ввода и правки надрубрик исключим поле надрубрики (super\_rubric), поскольку оно там совершенно не нужно и даже собьет с толку.

Листинг 4.4 показывает код классов редактора SuperRubricAdmin и встроенного редактора SubRubricInline. Помним, что код редакторов, обычных и встроенных, также как и код, регистрирующий модели и редакторы в подсистеме административного сайта, должен быть записан в модуль admin.py пакета приложения.

*Листинг 4.4. Код редактора SuperRubricAdmin и встроенного редактора SubRubricInline*

from .models import SuperRubric, SubRubric

class SubRubricInline(admin.TabularInline):

model = SubRubric

class SuperRubricAdmin(admin.ModelAdmin):

exclude = (**'super\_rubric'**,)

inlines = (SubRubricInline,)

admin.site.register(SuperRubric, SuperRubricAdmin)

Для работы с подрубриками понадобиться сделать поле надрубрики super\_rubric обязательным для заполнения. Для этого объявим форму SubRubricForm, записав ее код из листинга 4.5 в модуль forms.py пакета приложения.

*Листинг 4.5. Код формы SubRubricForm*

…

from .models import SuperRubric, SubRubric

…

class SubRubricForm(forms.ModelForm):

super\_rubric = forms.ModelChoiceField(

queryset=SuperRubric.object.all(), empty\_label=None,

label=**'Надрубрика'**, required=True

)

class Meta:

model = SubRubric

fields = **'\_\_all\_\_'**

У раскрывающегося списка, с помощью которого пользователь будет выбирать подрубрику, уберем “пустой” пункт, присвоив параметру empty\_label конструктора класса поля ModelChoiceField значение None. Так даем понять, что в это поле обязательно должно быть занесено значение.

Теперь можно написать код, объявляющий класс редактора SubRubricAdmin. Этот код представлен в листинге 4.6.

*Листинг 4.6. Код редактора SubRubricAdmin*

from .forms import SubRubricForm

class SubRubricAdmin(admin.ModelAdmin):

form = SubRubricForm

admin.site.register(SubRubric, SubRubricAdmin)

Сохраним весь исправленный код, запустим веб-сервер, войдем на административный сайт и добавим несколько надрубрик и подрубрик.

## Вывод рубрик в панели навигации

Осталось сделать так, чтобы в базовом шаблоне выводились пункты, представляющие все созданные к данному моменту рубрики.

Первое, что необходимо сделать, - поместить в состав контекста каждого шаблона переменную, в которой хранится список подрубрик. Можно создавать такую переменную в каждом контроллере, но это трудоемко. Поэтому объявим и зарегистрируем в проекте **обработчик контекста**, в котором и будет формироваться список подрубрик.

**Примечание:** Обработчик контекста - это программный модуль, добавляющий в контекст шаблона какие-либо дополнительные данные уже после его формирования контроллером. Обработчики контекста удобно использовать, если нужно просто добавить в контекст шаблона какие-либо данные.

Условимся, что список подрубрик будет помещаться в переменную rubrics контекста шаблона.

Создадим в пакете приложения модуль middlewares.py и запишем в него код обработчика контекста bboard\_context\_processor(), показанный в модуле 4.7.

*Листинг 4.7. Код обработчика контекста bboard\_context\_processor()*

from .models import SubRubric

def bboard\_context\_processor(request):

context = {}

context[**'rubrics'**] = SubRubric.object.all()

return context

Откроем модуль settings.py пакета конфигурации и зарегистрируем только что написанный обработчик контекста. Для этого добавим имя этого обработчика в список context\_processors из параметра OPTIONS:

TEMPLATES = [

{

**'BACKEND'**: **'django.template.backends.django.DjangoTemplates'**,

...

**'OPTIONS'**: {

**'context\_processors'**: [

...

**'main.middlewares.bboard\_context\_processor',**

],

},

},

]

Выполним еще несколько подготовительных действий. Во-первых, в модуле views.py пакета приложения объявим контроллер by\_rubric():

def by\_rubric(request, pk):

pass

Во-вторых, добавим в список маршрутов уровня приложения маршрут, ведущий на этот контроллер:

from .views import by\_rubric

**...**

urlpatterns = [

...

path(**'<int:pk>/'**, by\_rubric, name=**'by\_rubric'**),

path(**'<str:page>/'**, other\_page, name=**'other'**),

...

]

Объявление этого маршрута поместили перед объявление маршрута, ведущего на контроллер other\_page(), для того чтобы Django не принял код рубрики за шаблон страницы и не вызвал ошибку 404.

Контроллер-заглушку создали чтобы прямо сейчас сформировать в панели навигации гиперссылки с правильными адресами.

Откроем базовый шаблон и исправим код панели навигации следующим образом:

<nav class="col-md-auto nav flex-column">

<a class="nav-link root" href="{% url 'main:index' %}">Главная</a>

{% for rubric in rubrics %}

{% ifchanged rubric.super\_rubric.pk %}

<span class="nav-link root font-weight-bold">

{{ rubric.super\_rubric.name }}

</span>

{% endifchanged %}

<a class="nav-link" href="{% url 'main:by\_rubric' pk=rubric.pk %}">

{{ rubric.name }}

</a>

{% endfor %}

</nav>

В данном коде нет ничего сложного. В нем просто перебираем список подрубрик, хранящийся в переменной rubrics контекста шаблона и для каждой подрубрики выводим:

* если ключ связанной подрубрики изменился (те есть, если начали выводится подрубрики из другой надрубрики) - пункт с именем надрубрики;
* пункт-гиперссылку с именем подрубрики.

# 5. Объявления

Теперь можно приступать к работе над объявлениями. Создадим модель объявлений, модель дополнительных иллюстраций к ним, страницу для просмотра списка объявлений, относящихся к выбранной рубрике, с поддержкой пагинации и поиска, страницу сведений о выбранном объявлении, страницы для добавления, правки и удаления объявлений. Также сделаем так, чтобы на странице профиля выводился список объявлений, оставленных текущим пользователем.

## Обработка загруженных файлов

В составе объявлений будет присутствовать графическое изображение к продаваемому товару. Кроме того, пользователь может создать для объявления произвольное количество дополнительных иллюстраций, также представляющих собой изображения.

Чтобы Django смог обработать выгруженные посетителями файлы нужно установить три дополнительные библиотеки: Pillow (обеспечивает поддержку графики), Easy Thumbnails (создает миниатюры) и django-cleanup (удаляет загруженные файлы после удаления хранящих их записей моделей). Установим эти библиотеки:

**pip install pillow**

**pip install easy-thumbnails**

**pip install django-cleanup**

Добавим программные ядра двух последний библиотек в список зарегистрированных в проекте (файл settings.py пакета конфигурации):

INSTALLED\_APPS = [

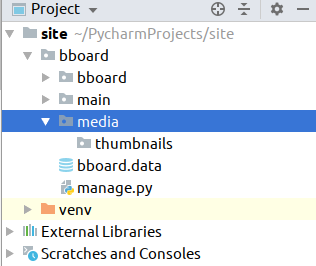
...

**'django\_cleanup'**,

**'easy\_thumbnails'**

]

Для хранения самих файлов создадим директорию media в директории проекта. Для хранения миниатюр создадим внутри директории media директорию thumbnails (Рис 5.1).



*Рис 5.1. Текущая структура директорий проекта*

В модуле settings.py укажум путь к директории media и префикс для интернет-адресов выгруженных файлов:

MEDIA\_ROOT = os.path.join(BASE\_DIR, **'media'**)

MEDIA\_URL = **'/media/'**

Туда же сразу добавим настройки приложения easy\_thumbnails:

THUMBNAIL\_ALIASES = {

**''**: {

**'default'**: {

**'size'**:(96, 96),

**'crop'**:**'scale'**,

},

},

}

THUMBNAIL\_BASEDIR = **'thumbnails'**

Для миниатюр был добавлен один-единственный пресет, используемый по умолчанию для всех случаев. Он указывает создавать миниатюры простым масштабированием до размеров 96x96 пикселов. Также было задано имя вложенной директории, в которой будут храниться миниатюры - thrmbnails.

Теперь нужно в модуле urls.py пакета конфигурации добавить код, создающий маршрут для обработки выгруженных файлов:

from django.conf.urls.static import static

urlpatterns = [

...

]

if settings.DEBUG:

urlpatterns.append(path(**'static/<path:path>'**, never\_cache(serve)))

urlpatterns += static(settings.MEDIA\_URL, document\_root = settings.MEDIA\_ROOT)

## Модели объявлений и дополнительных иллюстраций

Теперь создадим модели объявлений и дополнительных иллюстрации.

### Модель самих объявлений

Модель, хранящую объявления назовем Bb. Ее структура представлена в таблице ниже:

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Тип** | **Дополнительные параметры** | **Описание** |
| rubric | ForeignKey | Запрет каскадного удаления | Подрубрика |
| title | CharField | Длина - 40 символов | Название товара |
| content | TextField |  | Описание товара |
| price | IntegerField | Значение по умолчанию - 0 | Цена товара |
| contacts | TextField |  | Контакты |
| image | ImageField | Необязательное | Основная иллюстрация к объявлению |
| author | ForeignKey |  | Пользователь, оставивший объявление |
| is\_active | BooleanField | Значение по умолчанию - True, индексированное | Признак, показывать ли объявление в списке |
| created\_at | DateTimeField | Подставляется текущее значение даты и времени, индексированное | Дата и время публикации объявления |

В поле rubric, устанавливающем связь с моделью подрубрик SubRubric, указали запрет каскадного удаления, чтобы предотвратить случайное удаление подрубрики вместе со всеми оставленными в ней объявлениями.

Графические файлы, сохраняемые в поле image модели, будут иметь в качестве имен текущие отметки времени. Таким образом проведем имена к единому типу и заодно устраним ситуацию, когда выгруженный файл имеет настолько длинное имя, что оно не помещается в отведенное для хранения файла поле.

При разработке модели объявления нужно учесть еще один момент. Чуть позже напишем модель дополнительных иллюстраций, которую свяжем с моделью объявлений связью “один-ко-многим”. Если при объявлении этой связи разрешим каскадное удаление, то при удалении объявления будут удалены все относящиеся к ней дополнительные иллюстрации. Но это удаление выполнит не Django, а СУБД, отчего приложение django\_cleanup не получит сигнала об удалении записей и не сможет в ответ удалить хранящиеся в них графические файлы. В результате эти файлы остануться на диске бесполезным мусором.

Можно решить эту проблему, но для этого понадобиться условиться об имени модели дополнительных иллюстраций. Пусть эта модель называется AdditionalImage.

Ранее в пакете приложения был создан модуль utilities.py, в который записали объявление функции, выполняющей отправку писем. Этот модуль - подходящее место для сохранения кода, не относящегося напрямую ни к моделям, ни к редакторам, ни к контроллерам. Поместим в него объявление функции get\_timestamp\_path(), которая будет генерировать имена сохраняемых в модели выгруженных файлов (Листинг 5.1).

*Листинг 5.1. Код функции get\_timestamp\_path()*

from datetime import datetime

from os.path import splitext

def get\_timestamp\_path(instance, filename):

return **'%s%s'** % (datetime.now().timestamp(), splitext(filename)[1])

Теперь можно написать код, объявляющий сам класс модели Bb. Готовый код показан в листинге 5.2.

*Листинг 5.2. Код модели Bb*

from .utilities import get\_timestamp\_path

class Bb(models.Model):

rubric = models.ForeignKey(SubRubric, on\_delete=models.PROTECT, verbose\_name=**'Рубрика'**)

title = models.CharField(max\_length=40, verbose\_name=**'Товар'**)

content = models.TextField(verbose\_name=**'Описание'**)

price = models.FloatField(default=0, verbose\_name=**'Цена'**)

contacts = models.TextField(verbose\_name=**'Контакты'**)

image = models.ImageField(blank=True, upload\_to=get\_timestamp\_path, verbose\_name=**'Изображение'**)

author = models.ForeignKey(AdvUser, on\_delete=models.CASCADE, verbose\_name=**'Автор объявления'**)

is\_active = models.BooleanField(default=True, db\_index=True, verbose\_name=**'Выводить в списке?'**)

created\_at = models.DateTimeField(auto\_now\_add=True, db\_index=True, verbose\_name=**'Опубликовано'**)

def delete(self, \*args, \*\*kwargs):

for ai in self.additionalimage\_set.all():

ai.delete()

super().delete(\*args, \*\*kwargs)

class Meta:

verbose\_name\_plural = **'Объявления'**

verbose\_name = **'Объявление'**

ordering = [**'-created\_at'**]

Единственное, на что стоит обратить внимание - переопределенный метод delete(). В нем, перед тем как удалить текущую запись, перебираем и вызовом метода delete() удаляем все дополнительные иллюстрации, связанные с этой записью. При вызове метода delete() возникает сигнал post\_delete, обрабатываемый приложение django\_cleanup, которое в ответ удалит все файлы, хранящиеся в только что удаленной записи.

### Модель дополнительных иллюстраций

Модель дополнительных иллюстраций назовем AdditionalImage. Она гораздо проще предыдущей:

|  |  |  |
| --- | --- | --- |
| **Имя** | **Тип** | **Описание** |
| bb | ForeignKey | Объявление, к которому относится иллюстрация |
| image | ImageField | Собственно сама иллюстрация |

Графический файлы, сохраняемые в поле image этой модели, также в качестве имени получат текущие отметки времени. Для формирования имен файлов применим функцию get\_timestamp\_path() из модуля utilities.py.

Готовый код модели AdditionalImage можно увидеть в листинге 5.3.

*Листинг 5.3. Код модели AdditionalImage*

class AdditionalImage(models.Model):

bb = models.ForeignKey(Bb, on\_delete=models.CASCADE, verbose\_name=**'Объявление'**)

image = models.ImageField(upload\_to=get\_timestamp\_path, verbose\_name=**'Изображение'**)

class Meta:

verbose\_name\_plural = **'Дополнительные иллюстрации'**

verbose\_name = **'Дополнительная иллюстрация'**

Сохраним код моделей, произведем создание и выполнение миграций:

**python manage.py makemigrations**

**python manage.py migrate**

### Реализация удаления объявлений в модели пользователя

Ранее было сделано так, чтобы при удалении объявления явно удалялись все связанные с ним дополнительные иллюстрации. Это нужно для того, чтобы приложение django\_cleanup удалило все файлы, содержащие удаленные иллюстрации.

Сейчас нужно предусмотреть явное удаление объявлений, оставленных пользователем, при удалении этого пользователя. Для этого добавим в код модели AdvUser следующий фрагмент:

class AdvUser(AbstractUser):

...

def delete(self, \*args, \*\*kwargs):

for bb in self.bb\_set.all():

bb.delete()

super().delete(\*args, \*\*kwargs)

…

## Инструменты для администрирования объявлений

Для пользователей со статусом персонала, которые будут работать с объявлениями посредством административного сайта, создадим инструменты для администрирования объявлений. Объявим редактор BbAdmin, обеспечивающий работу с самими объявлениями,и встроенный редактор AdditionalImageInline, посредством которого пользователь сможет работать с дополнительными иллюстрациями непосредственно на страницах добавления и правки объявлений.

*Листинг 5.4. Код редактора BbAdmin и встроенного редактора AdditionalImageInline*

from .models import Bb, AdditionalImage

class AdditionalImageInline(admin.TabularInline):

model = AdditionalImage

class BbAdmin(admin.ModelAdmin):

list\_display = (**'rubric'**, **'title'**, **'content'**, **'author'**, **'created\_at'**)

fields = ((**'rubric'**, **'author'**), **'title'**, **'content'**, **'price'**,

**'contacts'**, **'image'**, **'is\_active'**)

inlines = (AdditionalImageInline,)

На страницах добавления и правки объявлений выведем раскрывающиеся списки подкатегории и пользователя в одну строку - ради компактности.

Сохраним код, запустим отладочный веб-сервер, войдем на административный сайт и добавим несколько объявлений, обязательно с дополнительными иллюстрациями. Попробуем исправить одно, удалить другое, проверив будут удалены ли файлы, хранящиеся в самом объявлении и связанные с ним иллюстрации.

## Вывод объявлений

Вывод списка объявлений довольно сложная задача. Для этого понадобится форма, в которую посетитель будет заносить искомое слово (конечно, форма не должна быть привязана к модели), контроллер и шаблон.

### Форма поиска и контроллер списка объявлений

Договоримся, что искомое слово, введенное посетителем, будем передавать контроллеру методом GET в GET-параметре с именем keyword. Следовательно, поле в форме для ввода искомого слова назовем точно так же.

Код формы поиска SearchForm очень прост:

*Листинг 5.5. Код формы SearchForm*

class SearchForm(forms.Form):

keyword = forms.CharField(required=False, max\_length=20, label=**''**)

Поскольку пользователь может ввести, а может и не ввести в keyword искомое слово, его сделали не обязательным. А еще убрали надпись у поля, присвоив параметру label пустую строку.

Ранее, чтобы вывести список рубрик, была создана ничего не делающая функция by\_rubric(). Перепишем ее так, чтобы код выглядел, как представлено в листинге 5.6.

*Листинг 5.6. Код контроллера-функции by\_rubric()*

from django.core.paginator import Paginator

from django.db.models import Q

from .forms import SearchForm

from .models import SubRubric, Bb

def by\_rubric(request, pk):

rubric = get\_object\_or\_404(SubRubric, pk=pk)

bbs = Bb.objects.filter(is\_active=True, rubric=pk)

if **'keyword'** in request.GET:

keyword = request.GET[**'keyword'**]

q = Q(title\_\_icontains=keyword) | Q(content\_\_icontains=keyword)

bbs = bbs.filter(q)

else:

keyword = **''**

form = SearchForm(initial={**'keyword'**: keyword})

paginator = Paginator(bbs, 2)

if **'page'** in request.GET:

page\_num = request.GET[**'page'**]

else:

page\_num = 1

page = paginator.get\_page(page\_num)

context = {**'rubric'**: rubric, **'page'**: page, **'bbs'**: page.object\_list, **'form'**: form}

return render(request, **'main/by\_rubric.html'**, context)

Сначала извлекаем выбранную посетителем рубрику, так как далее понадобиться вывести на странице ее название. Затем извлекаем объявления, относящиеся к этой рубрике и помеченные для вывода (те, у которых поле is\_active хранит значение True). После чего выполняем фильтрацию уже отобранных объявлений по введенному посетителем искомому слову. Это слово, как и условились, передается через GET-параметра keyword.

Ради простоты получаем искомое слово непосредственно через параметр GET. Затем формируем на основе полученного слова условие фильтрации, применив объект Q, и выполнив фильтрацию объявлений.

**Примечание:** Для формирования сложных условий фильтрации применяется класс Q из модуля django.db.models. Его конструктор записывается в следующем формате:

Q(<условие фильтрации>)

Условие фильтрации записывается в том же виде, что и в вызове метода filter(). Полученный экземпляр класса можно использовать в вызовах вместо условий фильтрации. Также можно объединить два экземпляра класса Q посредством операторов & и |, которые обозначают соответственно логическое И и ИЛИ.

Далее создаем экземпляр формы SearchForm, чтобы вывести ее на экран. Конструктору ее класса в параметре initial передаем полученное из GET-параметра keyword искомое слово - так что это слово будет присутствовать в выведенной на экран форме.

Кроме того создаем пагинатор, указав количество записей в одно его части равным 2. Это позволит проверить, работает ли пагинация, имея в базе всего несколько объявлений. Наконец, выводим страницу со списком объявлений, применив шаблон main/by\_rubric.html.

### Реализация корректного возврата

В существующей реализации присутствует небольшая проблема. Предположим, что посетитель заходит на сайт, выбирает какую-нибудь рубрику, пролистывает несколько частей, сформированных пагинатором, находит нужное объявление и щелкает на гиперссылке, чтобы просмотреть его содержимое. После просмотра объявления, он щелкает на гиперссылке для возврата на список объявлений …. И попадает на самую первую часть этого списка.

Как избежать проблемы, в общем, понятно. Номер выводимой части и искомое слово у передаются через GET-параметры с именами page и keyword соответственно. Тогда, чтобы вернуться на нужную часть списка уже отфильтрованных по заданному слову объявлений, нужно передать эти параметры странице сведений об объявлении.

По принятым в Django соглашениям, весь код, ответственный за формирование страниц, следует помещать в шаблон, посредник или в обработчик контекста (как раз наш случай).

Откроем модуль middlewares.py пакета приложения и найдем там обработчик контекста bboard\_context\_processor(), написанный нами ранее. Вставим в него фрагмент, который создаст в контексте шаблона две переменные:

* keyword - с полностью сформированным GET-параметром keyword, который понадобиться для генерирования интернет-адресов в гиперссылках навигатора;
* all - с полностью сформированными GET-параметрами keyword и page, которые добавим к интернет-адресам гиперссылок, указывающих на страницы сведений об объявлениях.

Добавленный код не использует никаких особо сложных приемов программирования:

def bboard\_context\_processor(request):

context = {}

context[**'rubrics'**] = SubRubric.object.all()

context[**'keyword'**] = **''**

context[**'all'**] = **''**

if **'keyword'** in request.GET:

keyword = request.GET[**'keyword'**]

if keyword:

context[**'keyword'**] = **'?keyword='** + keyword

context[**'all'**] = context[**'keyword'**]

if **'page'** in request.GET:

page = request.GET[**'page'**]

if page != **'1'**:

if context[**'all'**]:

context[**'all'**] += **'&page='** + page

else:

context[**'all'**] = **'?page='** + page

return context

### Шаблон страницы списка объявлений

Теперь можно приступать к написанию кода шаблона main/by\_rubric.html, который сформирует страницу объявлений. Код можно увидеть в листинге 5.7.

*Листинг 5.7. Код шаблона main/by\_rubric.html*

{% extends "layout/basic.html" %}

{% load thumbnail %}

{% load static %}

{% load bootstrap4 %}

{% block title %}{{ rubric }}{% endblock %}

{% block searchform %}

{% endblock %}

{% block content %}

<h2 class="mb-2">{{ rubric }}</h2>

<div class="container-fluid mb-2">

<div class="row">

<div class="col">&nbsp;</div>

<form class="col-md-auto form-inline">

{% bootstrap\_form form show\_label=False %}

{% bootstrap\_button content='Искать' button\_type='submit' %}

</form>

</div>

</div>

{% if bbs %}

<ul class="list-unstyled">

{% for bb in bbs %}

<li class="media my-5 p-3 border">

{% url 'main:detail' rubric\_pk=rubric.pk pk=bb.pk as url %}

<a href="{{ url }}{{ all }}">

{% if bb.image %}

<img class="mr-3" src="{% thumbnail bb.image 'default' %}">

{% else %}

<img class="mr-3" src="{% static 'main/empty.jpg' %}">

{% endif %}

</a>

<div class="media-body">

<h3><a href="{{ url }}{{ all }}">{{ bb.title }}</a> </h3>

<div>{{ bb.content }}</div>

<p class="text-right font-weight-bold">{{ bb.price }}</p>

<p class="text-right font-italic">{{ bb.created\_at }}</p>

</div>

</li>

{% endfor %}

</ul>

{% bootstrap\_pagination page url=keyword %}

{% endif %}

{% endblock %}

Чтобы вывести форму поиска, прижав ее к правой части страницы, используем конструкцию вида:

<div class="container-fluid mb-2">

<div class="row">

<div class="col">&nbsp;</div>

<form class="col-md-auto form-inline">

{% bootstrap\_form form show\_label=False %}

{% bootstrap\_button content='Искать' button\_type='submit' %}

</form>

</div>

</div>

Класс container-fluid заставляет элемент, к которому он привязан, вести себя как обычная таблица HTML. Элемент с привязанным стилевым классом row, вложенный в этот элемент, ведет себя как строка таблицы. Вложенный в элемент row элемент с классом col ведет себя как ячейка таблицы, растягивающаяся на всю ширину, а элемент с классом col-md-auto - как ячейка с шириной, равной ширине ее содержимого. Соответственно сама форма прижимается к правому краю.

Сам перечень создается маркированным списком с классом list-unstyled. Отдельная позиция в списке формируется пунктом списка <li>, к которому привязаны классы media - для удобной группировки изображения и текста, my-5 - большие внешние отступы сверху и снизу, p-3 - внутренние отступы среднего размера со всех сторон, border - рамку вокруг элемента. Внутрь помещается изображение с классом mr-3 и оставшиеся элементы.

Гиперссылки, указывающие на страницу об объявлении, были привязаны на основное изображение и название товара. Чтобы не генерировать адрес дважды, ссылка была сохранена в переменной url:

{% url 'main:detail' rubric\_pk=rubric.pk pk=bb.pk as url %}

Иллюстрация не является обязательной, потому в шаблоне предусмотрен случай когда ее нет - выводится изображение-заглушка.

### Вывод сведений о выбранном изображении

Сразу же создадим контроллер, маршрут и шаблон для страницы со сведениями о выбранном посетителем объявлении.

Начнем с маршрута. Поместим его перед маршрутом, ведущим на контроллер by\_rubric(), который выводит список объявлений:

from .views import detail

...

urlpatterns = [

...

path(**'<int:rubric\_pk>/<int:pk>/'**, detail, name=**'detail'**),

path(**'<int:pk>/'**, by\_rubric, name=**'by\_rubric'**),

...

]

Код контроллера detail() показан в листинге 5.8.

*Листинг 5.8. Код контроллера-функции detail()*

def detail(request, rubric\_pk, pk):

bb = get\_object\_or\_404(Bb, pk=pk)

ais = bb.additionalimage\_set.all()

context = {**'bb'**: bb, **'ais'**: ais}

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

Помимо самого объявления также подготовим список дополнительных иллюстраций к объявлению, записав его в переменную ais.

Код шаблона main/detail.html, выводящего страницу сведений об объявлении, можно увидеть ниже.

*Листинг 5.9. Код шаблона main/detail.html*

{% extends "layout/basic.html" %}

{% block title %}{{ bb.title }} - {{ bb.rubric.name }}{% endblock %}

{% block content %}

<div class="container-fluid mt-3">

<div class="row">

{% if bb.image %}

<div class="col-md-auto">

<img src="{{ bb.image.url }}" class="main-image">

</div>

{% endif %}

<div class="col">

<h2>{{ bb.title }}</h2>

<p>{{ bb.content }}</p>

<p class="font-weight-bold">{{ bb.price }} руб.</p>

<p>{{ bb.contacts }}</p>

<p class="text-right font-italic">Объявление добавлено {{ bb.created\_at }}</p>

</div>

</div>

</div>

{% if ais %}

<div class="d-flex justify-content-berween flex-wrap mt-5">

{% for ai in ais %}

<div>

<img src="{{ ai.image.url }}" class="additional-image">

</div>

{% endfor %}

</div>

{% endif %}

<p><a href="{% url 'main:by\_rubric' pk=bb.rubric.pk %}{{ all }}">Назад</a></p>

{% endblock %}

В принципе похожий код разметки уже использовали. Единственно заслуживает внимания код, что выводит дополнительные иллюстрации:

<div class="d-flex justify-content-berween flex-wrap mt-5">

{% for ai in ais %}

<div>

<img src="{{ ai.image.url }}" class="additional-image">

</div>

{% endfor %}

</div>

К блоку, в котором будут выводиться дополнительные иллюстрации, привязаны следующие классы:

* d-flex - устанавливает для элемента так называемую гибкую разметку, при которой дочерние элементы выстраиваются внутри родителя по горизонтали;
* justify-content-between - указывает, что дочерние элементы должны располагаться внутри на равномерном расстоянии друг от друга;
* flex-wrap - если дочерним элементам не хватит места, чтобы выстроиться по горизонтали, не помещающиеся элементы будут перенесены на следующую строку;
* mt-5 - большой внешний отступ сверху, чтобы отделить дополнительные иллюстрации от основной информации.

Кроме того с помощью классов main-image и additional-image были установлены размеры основной и дополнительных иллюстраций. К сожалению, данных классов нет в Bootstrap, потому придется открыть таблицу стилей main/style.css и добавить в нее фрагмент кода:

img.main-image {

width: 300px;

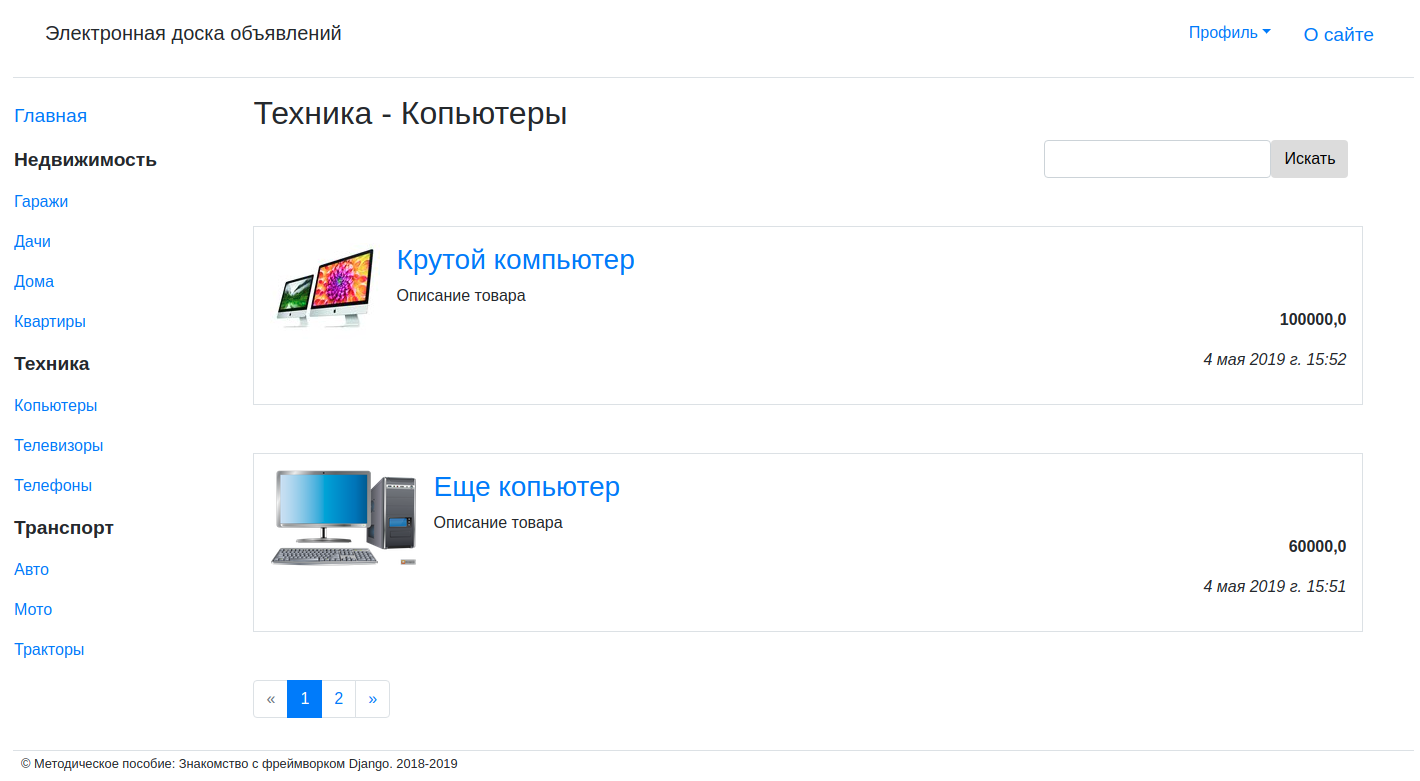
}

img.additional-image {

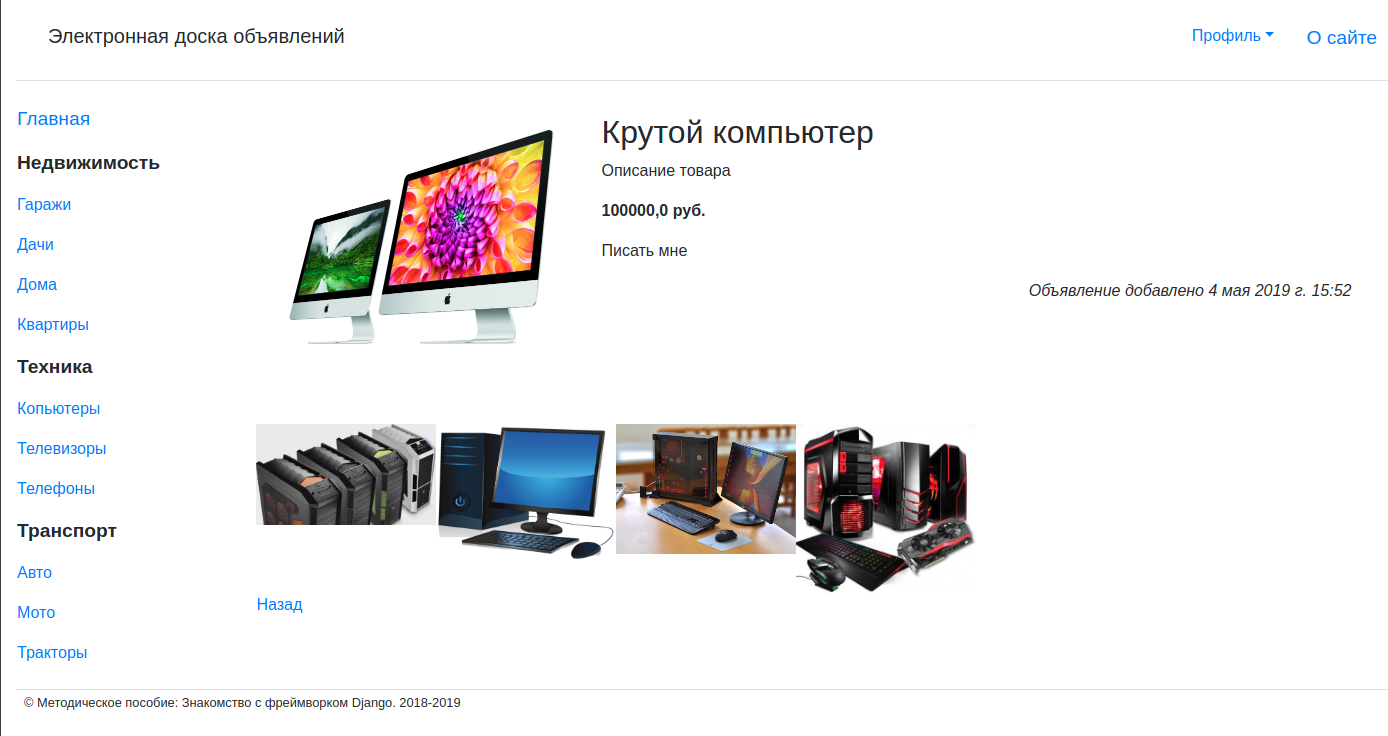
width: 180px;

}

Сохраним весь код, перезапустим отладочный сервер и зайдем на страницу объявлений относящихся к какой-либо из категорий. Если все сделано правильно, страница будет выглядеть так, как показано на рис. 5.2. После можно перейти на страницу со сведениями об объявлении (Рис. 5.3).



*Рис 5.2. Страница списка объявлений*



*Рис 5.3. Страница сведений об объявлении*

### Вывод последний 10 объявлений на главной странице

Сначала нужно найти код контроллера-функции index(), который выводит главную страницу, и добавить в него фрагмент, выбирающий из базы последние 10 объявлений:

def index(request):

bbs = Bb.objects.filter(is\_active=True)[:10]

context = {**'bbs'**:bbs}

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

После нужно в шаблоне main/index.html дописать код, который собственно выводит эти объявления.

**Примечание**: В качестве самостоятельного задания реализуйте вывод последних 10 объявлений на главной странице. За основу можно взять код из шаблона main/by\_rubric.html.

## Работа с объявлениями

### Вывод объявлений, оставленных текущим пользователем

Для вывода объявлений, написанных текущим пользователем нужно добавить в контроллер-функцию profile() следующий код:

@login\_required

def profile(request):

bbs = Bb.objects.filter(author=request.user.pk)

context = {**'bbs'**: bbs}

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

Далее в шаблоне main/profile.html нужно добавить код, который можно взять из шаблона main/by\_rubric.html практически без изменений.

**Примечание**: В качестве самостоятельного задания реализуйте вывод списка объявлений в профиле, написанных текущим пользователем.

### Добавление, правка и удаление объявлений

Чтобы реализовать добавление нового объявления, нужно сначала объявить форму для ввода самого объявления и набор форм, в которые будут заноситься дополнительные иллюстрации. Форма будет связана с моделью объявления Bb, а набор форм - с моделью дополнительной иллюстрации AdditionslImage (Листинг 5.10).

*Листинг 5.10. Код формы BbForm и набора форм AIFormSet*

from .models import Bb, AdditionalImage

from django.forms import inlineformset\_factory

class BbForm(forms.ModelForm):

class Meta:

model = Bb

fields = **'\_\_all\_\_'**

widgets = {**'author'**: forms.HiddenInput}

AIFormSet = inlineformset\_factory(Bb, AdditionalImage, fields=**'\_\_all\_\_'**)

В форме будут выводиться все поля модели Bb. Для поля автора объявления в качестве элемента управления было задано HiddenInput, т. е. скрытое поле, так как значение туда все равно будет заноситься автоматически.

Контроллер, добавляющий объявление, будет реализован в виде функции с названием profile\_bb\_add(). Код представлен в листинге 5.11.

*Листинг 5.11. Код контроллера-функции profile\_bb\_add()*

from django.shortcuts import refirect

from .forms import BbForm, AIFormSet

@login\_required

def profile\_bb\_add(request):

if request.method == **'POST'**:

form = BbForm(request.POST, request.FILES)

if form.is\_valid():

bb = form.save()

formset = AIFormSet(request.POST, request.FILES, instance=bb)

if formset.is\_valid():

formset.save()

messages.add\_message(request, messages.SUCCESS,

**'Объявление добавлено'**)

return redirect(**'main:profile'**)

else:

form = BbForm(initial={**'author'**: request.user.pk})

formset = AIFormSet()

context = {**'form'**: form, **'formset'**: formset}

return render(request, **'main/profile\_bb\_add.html'**, context)

Здесь нужно отметить три важных момента. Во-первых, при создании формы перед выводом страницы в поле author формы заносится идентификатор текущего пользователя. Таким образом связывается заносимое объявление с этим пользователем.

Во-вторых, во время сохранения введенного объявления, при создании объектов формы и набора форм, необходимо передавать конструкторам их классов вторым позиционным параметром словарь со всеми полученными файлами (атрибут FILES объекта запроса). Если не сделать этого, отправленные пользователем иллюстрации окажутся потерянными.

В-третьих, при сохранении сначала выполняем вализацию и сохранение формы самого объявления. Метод save() в качестве результата возвращает сохраненную запись и эту запись нужно передать конструктору класса набора форм через параметр instance. Это нужно для того, чтобы все дополнительные иллюстрации после сохранения оказались связанными с объявлениями.

Теперь нужно написать маршрут, который укажет на страницу добавления объявления, поместив его перед маршрутом, указывающим на страницу профиля:

from .views import profile\_bb\_add

...

urlpatterns = [

...

path(**'accounts/profile/add/'**, profile\_bb\_add, name=**'profile\_bb\_add'**),

path(**'accounts/profile/'**, profile, name=**'profile'**),

...

]

Осталось написать шаблон main/profile\_bb\_add.html, который создаст страницу добавления объявления. Его код можно увидеть в листинге 5.12.

*Листинг 5.12. Код шаблона main/profile\_bb\_add.html*

{% extends "layout/basic.html" %}

{% load bootstrap4 %}

{% block title %} Добавление объявления {% endblock %}

{% block content %}

<h2>Добавление объявления</h2>

<form method="post" enctype="multipart/form-data">

{% csrf\_token %}

{% bootstrap\_form form layout='horizontal' %}

{% bootstrap\_formset formset layout='horizontal' %}

{% buttons submit='Добавить' %} {% endbuttons %}

</form>

{% endblock %}

Обязательно указать у формы метод кодирования данных multipart/form-data. Если этого не будет сделано, то файлы не будут отправлены. Набор форм добавляется с помощью тега шаблонизатора bootstrap\_formset.

Наконец, в шаблон профиля main/profile.html добавим гиперссылку, указывающую на страницу добавления объявления:

<p><a href="{% url 'main:profile\_bb\_add' %}">Добавить объявление</a> </p>

Контроллеры proflle\_bb\_change() и profile\_bb\_delete(), которые соответсвенно правят и удаляют объявление, написать не сложно. Их код представлен в листинге 5.13.

*Листинг 5.13. Код контроллеров-функций profile\_bb\_change() и profile\_bb\_delete()*

@login\_required

def profile\_bb\_change(request, pk):

bb = get\_object\_or\_404(Bb, pk=pk)

if not request.user.is\_author(bb):

return redirect(**'main:profile'**)

if request.method == **'POST'**:

form = BbForm(request.POST, request.FILES, instance=bb)

if form.is\_valid():

bb = form.save()

formset = AIFormSet(request.POST, request.FILES, instance=bb)

if formset.is\_valid():

formset.save()

messages.add\_message(request, messages.SUCCESS,

**'Объявление изменено'**)

return redirect(**'main:profile'**)

else:

form = BbForm(instance=bb)

formset = AIFormSet(instance=bb)

context = {**'form'**: form, **'formset'**: formset}

return render(request, **'main/profile\_bb\_change.html'**, context)

@login\_required

def profile\_bb\_delete(request, pk):

bb = get\_object\_or\_404(Bb, pk=pk)

if not request.user.is\_author(bb):

return redirect(**'main:profile'**)

if request.method == **'POST'**:

bb.delete()

messages.add\_message(request, messages.SUCCESS,

**'Объявление удалено'**)

return redirect(**'main:profile'**)

else:

context = {**'bb'**: bb}

return render(request, **'main/profile\_bb\_delete.html'**, context)

Для проверки на авторство объявления был использован такой фрагмент кода:

if not request.user.is\_author(bb):

return redirect(**'main:profile'**)

Метод is\_author() модели пользователя нужно добавить в соответствующий класс:

class AdvUser(AbstractUser):

...

def is\_author(self, bb):

if self.pk == bb.author.pk:

return True

return False

class Meta(AbstractUser.Meta):

pass

Он проверяет соответствие идентификаторов авторизованного пользователя и автора текущего объявления.

Объявим необходимые маршруты:

from .views import profile\_bb\_delete

...

urlpatterns = [

...

path(**'accounts/profile/change/<int:pk>/'**, profile\_bb\_change, name=**'profile\_bb\_change'**),

path(**'accounts/profile/delete/<int:pk>/'**, profile\_bb\_delete, name=**'profile\_bb\_delete'**),

path(**'accounts/profile/'**, profile, name=**'profile'**),

...

]

**Примечание:** В качестве самостоятельного задания напишите необходимые шаблоны main/profile\_bb\_change.html и main/profile\_bb\_delete.html, использовав ранее написанные шаблоны. Кроме того в шаблон main/profile.html добавьте код, создающий гиперссылки для правки и удаления каждого из добавленных пользователем объявления.

# 6. Комментарии

К каждому из опубликованных на нашем сайте объявлений посетители могут оставить комментарии.

## Подготовка к выводу CAPTCHA

Сделаем так, чтобы зарегистрированные пользователи могли оставлять комментарии беспрепятственно, а гости должны были бы дополнительно занести в форму CAPTCHA. Необходимо предварительно установить библиотеку Django Simple Captcha:

**pip install django-simple-captcha**

Добавим в список зарегистрированных в проекте приложение captcha - программное ядро этой библиотеки:

INSTALLED\_APPS = [

...

**'captcha'**,

]

И объявим в списке маршрутов уровня проекта маршрут, указывающий на это приложение:

urlpatterns = [

...

path(**'captcha'**, include(**'captcha.urls'**)),

path(**''**, include(**'main.urls'**, namespace=**''**)),

]

Миграции пока выполнять не будем, так как все равно еще нужно будет объявить модель для комментариев.

## Модель комментария

Модель комментариев назовем Comment. Ее структура приведена в таблице ниже:

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Тип** | **Дополнительные параметры** | **Описание** |
| bb | ForeignKey | Каскадное удаление разрешено | Объявление, к которому оставлен комментарий |
| author | CharField | Длина - 30 символов | Имя автора |
| content | TextField |  | Содержание |
| is\_active | BooleanField | Значение по умолчанию - True, индексированное | Признак, показывать ли комментарий в списке |
| created\_at | DateTimeField | Подставляется текущее значение даты и времени, индексированное | Дата и время публикации комментария |

Код, объявляющий модель Comment, приведен в листинге 6.1.

*Листинг 6.1. Код модели Comment*

class Comment(models.Model):

bb = models.ForeignKey(Bb, on\_delete=models.CASCADE,

verbose\_name=**'Объявление'**)

author = models.CharField(max\_length=30, verbose\_name=**'Автор'**)

content = models.TextField(verbose\_name=**'Содержание'**)

is\_active = models.BooleanField(default=True, db\_index=True,

verbose\_name=**'Выводить на экран?'**)

created\_at = models.DateTimeField(auto\_now\_add=True, db\_index=True,

verbose\_name=**'Опубликован'**)

class Meta:

verbose\_name\_plural = **'Комментарии'**

verbose\_name = **'Комментарий'**

ordering = [**'-created\_at'**]

Создадим и выполним все миграции:

**python manage.py makemigrations**

**python manage.py migrate**

## Вывод и добавление комментариев

Выводить сам список комментариев и форму добавления комментария будем на общедоступной странице сведений об объявлении.

Сначала нужно объявить две формы, в которые будут вносить комментарии, соответственно, зарегистрированные пользователи, выполнившие вход и и гости. Код форм UserCommentForm и GuestCommentForm приведен в листинге 6.2.

*Листинг 6.2. Код форм UserCommentForm и GuestCommentForm*

from captcha.fields import CaptchaField

from .models import Comment

class UserCommentForm(forms.ModelForm):

class Meta:

model = Comment

exclude = (**'is\_active'**,)

widgets = {**'bb'**: forms.HiddenInput}

class GuestCommentForm(forms.ModelForm):

captcha = CaptchaField(label=**'Введиде текст с картинки'**,

error\_messages={**'invalid'**: **'Неправильный текст'**})

class Meta:

model = Comment

exclude = (**'is\_active'**,)

widgets = {**'bb'**: forms.HiddenInput}

Теперь необходимо существенно обновить код контроллера функции detail(). В этом контроллере будет реализован и вывод уже добавленных к объявлению комментариев, и добавление нового. Код обновленного контроллера detail() представлен в листинге 6.3.

*Листинг 6.3. Код обновленного контроллера-функции detail()*

def detail(request, rubric\_pk, pk):

bb = get\_object\_or\_404(Bb, pk=pk)

ais = bb.additionalimage\_set.all()

comments = Comment.objects.filter(bb=pk, is\_active=True)

initial = {**'bb'**: bb.pk}

if request.user.is\_authenticated:

initial[**'author'**] = request.user.username

form\_class = UserCommentForm

else:

form\_class = GuestCommentForm

form = form\_class(initial=initial)

if request.method == **'POST'**:

c\_form = form\_class(request.POST)

if c\_form.is\_valid():

c\_form.save()

messages.add\_message(request, messages.SUCCESS,

**'Комментарий добавлен'**)

else:

form = c\_form

messages.add\_message(request, messages.WARNING,

**'Комментарий не добавлен'**)

context = {**'bb'**: bb, **'ais'**: ais, **'comments'**: comments, **'form'**: form}

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

Сразу после того, как будет подготовлено объявление, которое следует вывести на странице, список дополнительных иллюстраций к ней и список оставленных под ним комментариев, возникает задача создать форму для добавления комментария. При этом если текущий пользователь выполнил вход на сайт, нам нужно занести его имя в поле author формы комментария, чтобы пользователю было удобнеею Далее, в любом случае нужно занести в поле bb формы идентификатор текущего объявления. Наконец, если текущий пользователь выполнил вход на сайт, форма должна создаваться на основе класса UserCommentForm, а если не выполнил - на основе класса GuestCommentUser. Все эти действия выполняет фрагмент кода:

initial = {**'bb'**: bb.pk}

if request.user.is\_authenticated:

initial[**'author'**] = request.user.username

form\_class = UserCommentForm

else:

form\_class = GuestCommentForm

form = form\_class(initial=initial)

Объект формы был сохранен в переменной form. Форме из этой переменной впоследствии будет выведена на странице сведений об объявлении.

Далее, если полученный запрос был отправлен методом POST, то есть пользователь ввел комментарий о отправил его на сохранение, мы создаем еще один объект формы, передав конструктору полученные данные. Второй объект формы сохраняется в переменной c\_form. После этого мы выполняем валидацию второй формы.

Если валидация прошла неуспешно, мы переносим форму из переменной c\_form в переменную form. Эта форма, хранящая некорректные данные и сообщения об ошибках ввода, впоследствии будет выведена на экран, так что посетитель сразу сможет увидеть, что он ввел не так. Кроме того, выведем всплывающее сообщение о неуспешном добавлении комментария.

В шаблоне main/detail.html отыщем тег шаблонизатора block content … endblock и вставим непосредственно перед закрывающим тегом следующий код, который и выведет комментарии:

<h4 class="mt-5">Новый комментарий</h4>

<form method="post">

{% csrf\_token %}

{% bootstrap\_form form layout='horizontal' %}

{% buttons submit='Добавить' %}{% endbuttons %}

</form>

{% if comments %}

<div class="mt-5">

{% for comment in comments %}

<div class="my-2 p-2 border">

<h5>{{ comment.author }}</h5>

<p>{{ comment.content }}</p>

<p class="text-rigth font-italic">{{ comment.created\_at }}</p>

</div>

{% endfor %}

</div>

{% endif %}

В начале шаблона также необходимо добавить:

{% extends "layout/basic.html" %}

{% load bootstrap4 %}

…

**Примечание**: В качестве самостоятельного задания добавьте список комментариев на административную страницу сведений об объявлении. Добавьте также возможность модерации комментариев из административного сайта Django.

## Отправка уведомлений о появлении новых комментариев

Для отправки уведомлений о появлении новых комментариев будем обрабатывать сигнал post\_save, возникающий после сохранения записи в модель. Причем обрабатывать будем только сигналы, возникающие при сохранении записи модели Comment.

Откроем модуль utilities.py пакета приложения и добавим в него объявление функции send\_new\_comment\_notification(), которая и выполнит отправку уведомления. Ее код похож на код аналогичной функции send\_activation\_notification() и приведен в листинге 6.4.

*Листинг 6.4. Код функции send\_new\_comment\_notification()*

def send\_new\_comment\_notification(comment):

if ALLOWED\_HOSTS:

host = **'http://'** + ALLOWED\_HOSTS[0]

else:

host = **'http://localhost:8000'**

author = comment.bb.author

context = {**'author'**: author, **'host'**: host, **'comment'**: comment}

subject = render\_to\_string(**'email/new\_comment\_letter\_subject.txt'**,

context)

body\_text = render\_to\_string(**'email/new\_comment\_letter\_body.txt'**,

context)

author.email\_user(subject, body\_text)

**Примечание**: В качестве самостоятельного задания создайте шаблоны email/new\_comment\_letter\_subject.txt и email/new\_comment\_letter\_body.txt.

Осталось лишь привязать к сигналу post\_save обработчик, который будет вызывать функцию send\_new\_comment\_notification() после добавления комментария. Код выполняющий привязку, поместим в модуль models.py пакета приложения. Вот он:

from django.db.models.signals import post\_save

def post\_save\_dispatcher(sender, \*\*kwargs):

author = kwargs[**'instance'**].bb.author

if kwargs[**'created'**] and author.send\_messages:

send\_new\_comment\_notification(kwargs[**'instance'**])

post\_save.connect(post\_save\_dispatcher, sender=Comment)

# 7. Веб-служба REST

В данном разделе будет написана веб-служба, работающая по принципам REST. Она будет выдавать список из 10 последних объявлений, сведения о выбранном объявлении, список комментариев, оставленных для выбранного объявления, а также даст посетителям возможность добавлять новые комментарии.

**Примечание**: REST (Representational State Transfer, репрезентативная передача состояния) - серверная архитектура, следующая определенным принципам. Е числу таких принципов, в частности относится идентификация запрашиваемых ресурсов посредством обычных интернет-адресов. Веб-сайт, предоставляющий программные интерфейсы согласно принципам REST выступает в качестве сервера. Он отсылает клиентам данные, закодированные в каком-либо компактном формате, обычно это JSON. Клиенты получают эти данные и обрабатывают нужным им образом:выводят на экран, сохраняют в локальной базе данных и др.

## Предварительная настройка

Заниматься программированием веб-службы будем с применением библиотеки Django REST framework.

## Подготовка к разработке веб-службы

Сначала нужно установить библиотеки Django REST framework и django-cors-headers:

**pip install djangorestframework**

**pip install django-cors-headers**

Сразу же создадим новое приложение api, в котором и реализуем функциональность веб-службы:

**python manage.py startapp api**

Добавим приложение rest\_framework и corsheaders - программные ядра только что установленных библиотек, - а также только что созданное приложение api в список зарегистрированных в проекте:

INSTALLED\_APPS = [

...

**'rest\_framework'**,

**'corsheaders'**,

**'api.apps.ApiConfig'**,

]

Добавим в список зарегистрированных в проекте необходимый для работы посредник (порядок имеет значение):

MIDDLEWARE = [

..

**'corsheaders.middleware.CorsMiddleware'**,

**'django.middleware.common.CommonMiddleware'**,

...

]

Также в том же модуле settings.py пакета конфигурации, укажем настройки, разрешающие доступ к веб-службе с любого домена:

CORS\_ORIGIN\_ALLOW\_ALL = True

CORS\_URLS\_REGEX = **r'^/api/.\*S'**

## Список объявлений

Создадим в пакете приложения api модуль serializers.py. В нем мы запишем код нашего первого сериализатора, который будет формировать список рубрик и который назовем BbStrializer (Листинг 7.1).

**Примечание**: Сериализатор в Django REST framework выступает в той же роли, что и форма в Django. Он представляет извлеченные из базы сайта данные стороннему приложению, которое запрашивает их посредством обращения к программному интерфейсу. Сериализатор, как и форма, содержит объявления полей, либо записанные явно, либо созданные на основе полей связанной с ним модели.

*Листинг 7.1. Код сериализатора BbSerializer*

from rest\_framework import serializers

from main.models import Bb

class BbSerializer(serializers.ModelSerializer):

class Meta:

model = Bb

fields = (**'id'**, **'title'**, **'content'**, **'price'**, **'created\_at'**)

Контроллер, который будет выдавать список объявлений, реализуем в виде функции и назовем bbs(). Его код показан в листинге 7.2.

*Листинг 7.1. Код контроллера-функции bbs()*

from rest\_framework.response import Response

from rest\_framework.decorators import api\_view

from main.models import Bb

from .serializers import BbSerializer

@api\_view([**'GET'**])

def bbs(request):

if request.method == **'GET'**:

bbs = Bb.objects.filter(is\_active=True)[:10]

serializer = BbSerializer(bbs, many=True)

return Response(serializer.data)

Откроем список маршрутов уровня проекта (модуль urls.py пакета конфигурации) и добавим маршрут, указывающий на приложение api:

urlpatterns = [

...

path(**'api/'**, include(**'api.urls'**)),

path(**''**, include(**'main.urls'**, namespace=**''**)),

]

В пакете приложения api создадим файл urls.py, в который запишем список маршрутов уровня приложения. Код этого модуля представлен в листинге 7.3.

*Листинг 7.3. Код модуля urls.py пакета приложения api*

from django.urls import path

from .views import bbs

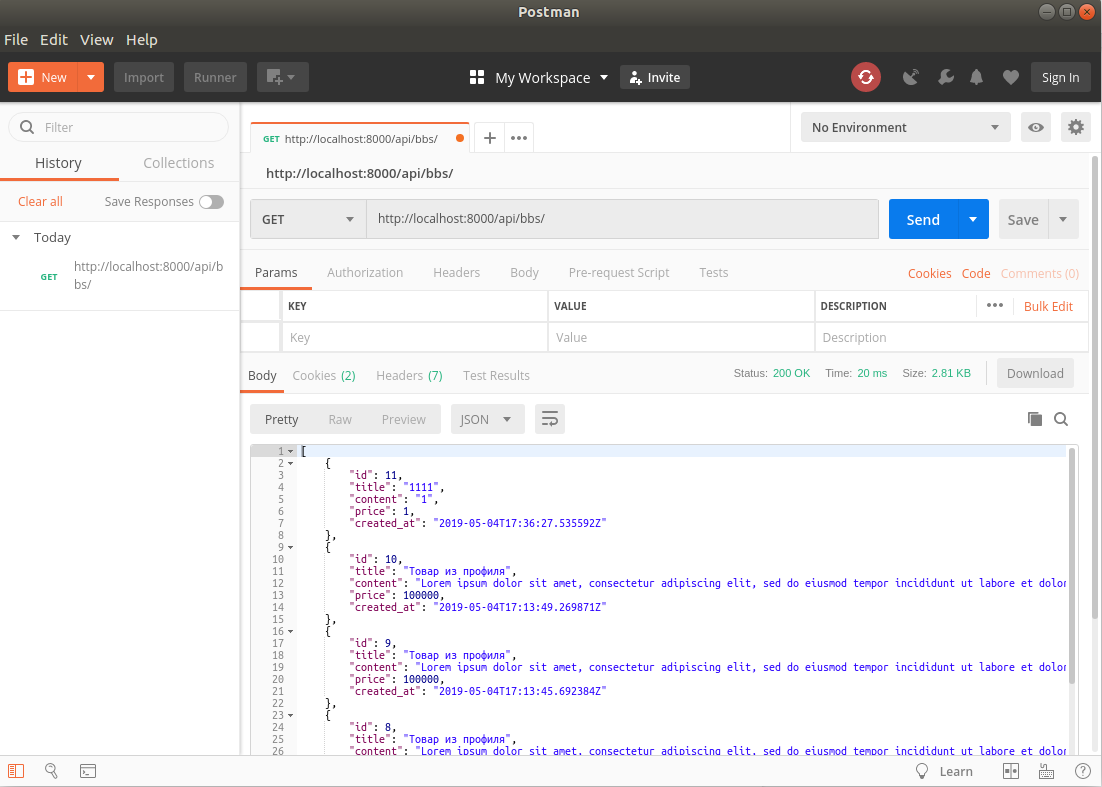
urlpatterns = [

path(**'bbs/'**, bbs),

]

Сохраним код, запустим отладочный веб-сервер и попробуем получить список объявлений, перейдя по интернет-адресу <http://localhost:8000/api/bbs/>. Если все было сделано правильно, то на странице отобразиться веб-представление, показывающее последние 10 объявлений, что было оставлены посетителями сайта.

**Примечание**: Для удобства работы с REST можно использовать бесплатную программу POSTMAN. В ней можно отправлять запросы к серверу разными методами, включая нужные заголовки, тело запроса и прочее. Интерфейс программы можно посмотреть на рис. 7.1.



*Рис 7.1. Интерфейс программы Postman*

## Сведения о выбранном объявлений

Напишем сериализатор BbDetailSerializer для отображения сведений о выбранном объявлении. Код представлен в листинге 7.4.

*Листинг 7.4. Код сериализатора BbDetailSerializer*

class BbDetailSerializer(serializers.ModelSerializer):

class Meta:

model = Bb

fields = (**'id'**, **'title'**, **'content'**, **'price'**, **'created\_at'**,

**'contacts'**, **'image'**)

Контроллер, который будет выдавать клиентам сведения о выбранном объявлении, реализуем в виде класса, производного от класса RetrieveAPIVies, который выполнит за почти всю работу. Код контроллера-класса представлен в листинге 7.5.

*Листинг 7.5. Код контроллера-класса BbDetailView*

from rest\_framework.generics import RetrieveAPIView

from .serializers BbDetailSerializer

class BbDetailView(RetrieveAPIView):

queryset = Bb.objects.filter(is\_active=True)

serializer\_class = BbDetailSerializer

Добавим в список маршрутов приложения маршрут, который укажет на новый контроллер:

from .views import bbs, BbDetailView

urlpatterns = [

path(**'bbs/<int:pk>/'**, BbDetailView.as\_view()),

path(**'bbs/'**, bbs),

]

Сохраним код и попытаемся получить сведения об объявлении с идентификатором 1, для чего нужно отправить GET запрос на адрес <http://localhost:8000/api/bbs/1>.

## Вывод и добавление комментариев

Теперь займемся функционалом для извлечения списка комментариев, оставленных под выбранным объявлением, и добавления новых комментариев.

Код сериализатора CommentSerializer, который будет отправлять список комментариев и добавлять новый комментарий можно увидеть в листинге 7.6.

*Листинг 7.6. Код сериализатора CommentSerializer*

from main.models import Comment

class CommentSerializer(serializers.ModelSerializer):

class Meta:

model = Comment

fields = (**'bb'**, **'author'**, **'content'**, **'created\_at'**)

Код контроллера-функции comments(), выдающего список комментариев и добавляющий новый комментарий, показан в листинге 7.7.

*Листинг 7.7. Код контроллера-функции comments()*

from rest\_framework.decorators import permission\_classes

from rest\_framework.status import HTTP\_201\_CREATED, HTTP\_400\_BAD\_REQUEST

from rest\_framework.permissions import IsAuthenticatedOrReadOnly

from main.models import Comment

from .serializers import CommentSerializer

@api\_view([**'GET'**, **'POST'**])

@permission\_classes((IsAuthenticatedOrReadOnly,))

def comments(request, pk):

if request.method == **'POST'**:

serializer = CommentSerializer(data=request.data)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data, status=HTTP\_201\_CREATED)

else:

return Response(serializer.errors, status=HTTP\_400\_BAD\_REQUEST)

else:

comments = Comment.objects.filter(is\_active=True, bb=pk)

serializer = CommentSerializer(comments, many=True)

return Response(serializer.data)

Контроллер был помечен декоратором IsAuthenticatedOrReadOnly, так как только зарегистрированные пользователи могут добавлять комментарии, а остальные только читать. В остальном в коде нет ничего сложного.

Добавим маршрут, который укажет на новый контроллер:

from .views import comments

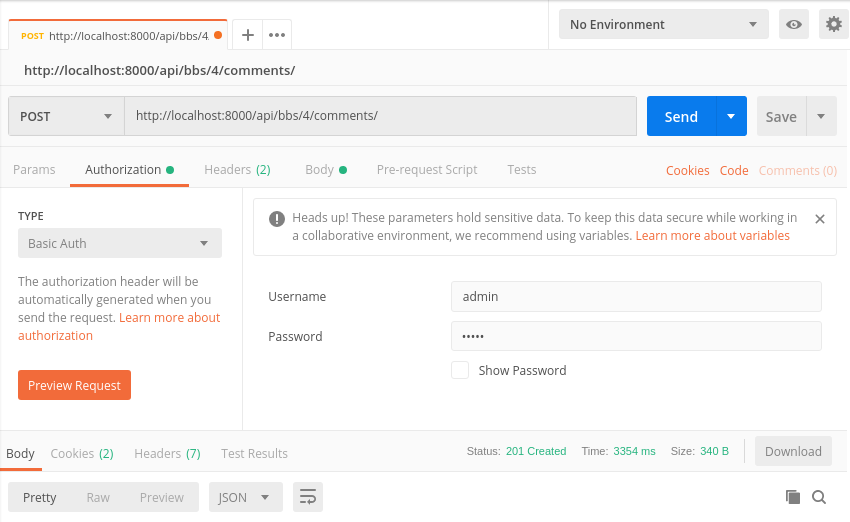
urlpatterns = [

path(**'bbs/<int:pk>/comments/'**, comments),

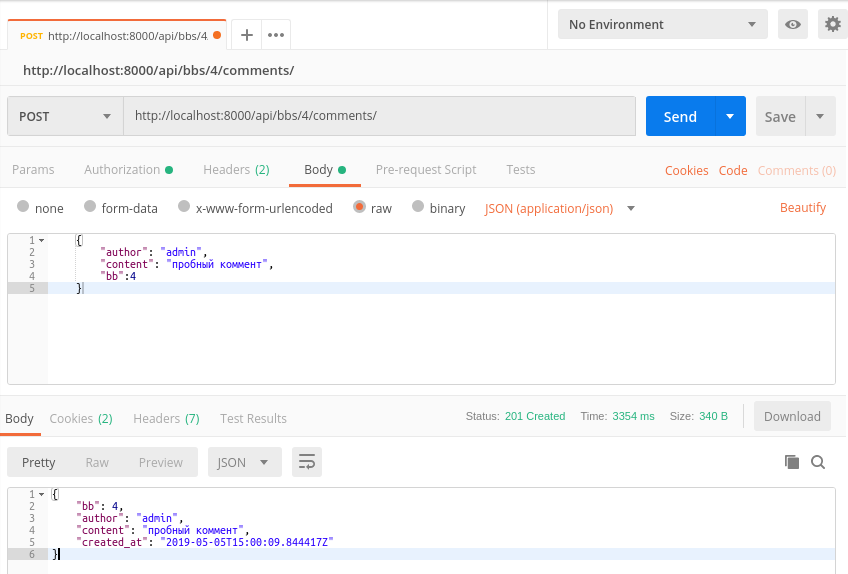
...

]

Проверим работоспособность вывода списка комментариев к определенному объявлению, для чего нужно отправить GET запрос на адрес <http://localhost:8000/api/bbs/1/comments/> Для того, чтобы проверить работоспособность добавления нового комментария, необходимо на этот же адрес отправить POST запрос и в теле запросе передать JSON объект. Кроме того, нужно передать параметры, необходимые для авторизации (рис. 7.2-7.3)



*Рис. 7.2. Отправка POST запроса для создания комментария - данные авторизации*



*Рис. 7.3. Отправка POST запроса для создания комментария - ответ сервера*