4

Разработка социального веб-сайта

В предыдущей главе вы научились реализовывать систему тегирования и рекомендовать схожие посты. Вы реализовали конкретно-прикладные шаблонные теги и фильтры. Вы также научились создавать карты сайта и новостные ленты для сайта и разработали полнотекстовый поисковый механизм с использованием базы данных PostgreSQL.

В этой главе вы научитесь разрабатывать функциональности с использованием учетной записи пользователя с целью создания социального сайта, включая регистрацию пользователя, управление паролем, редактирование профиля и аутентификацию. В последующих главах на этом сайте будут реализованы социальные функциональности, позволяющие пользователям делиться изображениями и взаимодействовать друг с другом. Пользователи смогут делать закладку на любое изображение в интернете и делиться им с другими пользователями. Они также смогут видеть внутриплатформенную активность пользователей, на которых они подписаны, и отмечать понравившиеся / не понравившиеся им изображения.

В этой главе будут рассмотрены следующие темы:

- создание представления входа в систему;
- использование встроенного в Django фреймворка аутентификации;
- создание шаблонов представлений входа в систему и выхода из системы, смены и сброса пароля;
- расширение модели пользователя с помощью конкретно-прикладной модели профиля;
- создание представлений регистрации пользователей;
- конфигурирование проекта под закачивание медиафайлов на сайт;
- использование фреймворка сообщений;
- разработка конкретно-прикладного бэкенда аутентификации;
- запрет на использование пользователями существующей электронной почты.

Давайте начнем с создания нового проекта.

Исходный код к этой главе находится по адресу https://github.com/PacktPublishing/Django-4-by-example/tree/main/Chapter04.

Все используемые в данной главе пакеты Python включены в файл requirements.txt в исходном коде к этой главе. Следуйте инструкциям по установке каждого пакета Python в последующих разделах либо установите требующиеся пакеты сразу с помощью команды `pip install -r requirements.txt`.

Создание проекта социального веб-сайта

Мы собираемся создать социальное приложение, и оно позволит пользователям делиться изображениями, которые они находят в интернете. В этом проекте нам потребуется разработать следующие элементы:

- система аутентификации, позволяющая пользователям регистрироваться, входить в систему, редактировать свой профиль, менять или сбрасывать пароль;
- система подписки, позволяющая пользователям подписываться друг на друга на сайте;
- функциональность отображения выложенных изображений и система, позволяющая пользователям делиться изображениями с любого веб-сайта;
- поток активности, который позволяет пользователям видеть контент, закачанный на сайт людьми, на которых они подписаны.

В этой главе будет рассмотрен первый пункт данного списка.

Запуск проекта социального веб-сайта

Откройте терминал и примените следующие ниже команды, чтобы создать виртуальную среду проекта:

```
mkdir env
python -m venv env/bookmarks
```

Если вы используете Linux или macOS, то для активации виртуальной среды выполните следующую ниже команду:

```
source env/bookmarks/bin/activate
```

Если вы используете Windows, то вместо этого примените следующую ниже команду:

```
.\env\bookmarks\Scripts\activate
```

В командной оболочке будет отображена активная виртуальная среда, как показано ниже:

```
(bookmarks)laptop:~ zenx$
```

Следующей ниже командой установите Django в своей виртуальной среде:

```
pip install Django~=4.1.0
```

Выполните следующую ниже команду, чтобы создать новый проект:

```
django-admin startproject bookmarks
```

Первоначальная структура проекта создана. С помощью следующих ниже команд войдите в каталог проекта и создайте новое приложение с именем `account`:

```
cd bookmarks/
django-admin startapp account
```

Напомним, что новое приложение добавляется в проект путем добавления имени приложения в настроечный параметр `INSTALLED_APPS` файла `settings.py`.

Отредактируйте файл `settings.py`, добавив в список `INSTALLED_APPS` следующую ниже строку, выделенную жирным шрифтом, перед всеми другими установленными приложениями:

```
INSTALLED_APPS = [
    'account.apps.AccountConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Django просматривает шаблоны в каталогах `template` приложений в порядке их появления в настроечном параметре `INSTALLED_APPS`. Приложение `django.contrib.admin` содержит стандартные шаблоны аутентификации, которые мы будем переопределять в приложении `account`. Помещая приложение первым в настроечном параметре `INSTALLED_APPS`, мы обеспечиваем, что по умолчанию будут использоваться конкретно-прикладные шаблоны аутентификации, а не шаблоны аутентификации, содержащиеся в `django.contrib.admin`.

Выполните следующую ниже команду, чтобы синхронизировать базу данных с моделями стандартных приложений, включенных в настроечный параметр `INSTALLED_APPS`:

```
python manage.py migrate
```

Вы увидите, что будут применены все изначальные миграции базы данных Django. Далее в проект будет встроена система аутентификации, используя поставляемый с Django фреймворк аутентификации.

Использование поставляемого с Django фреймворка аутентификации

Django поставляется вместе со встроенным фреймворком аутентификации, который способен оперировать аутентификацией пользователей, сеансами, разрешениями и группами пользователей. Система аутентификации содержит представления обычных действий пользователя, таких как вход в систему, выход из системы, смена пароля и сброс пароля.

Фреймворк аутентификации находится в приложении `django.contrib.auth` и используется другими пакетами Django `contrib`. Напомним, что мы уже применяли фреймворк аутентификации в *главе 1 «Разработка приложения для ведения блога»*, чтобы создать суперпользователя для приложения `blog` с целью доступа к сайту администрирования.

При создании нового проекта Django с помощью команды `startproject` фреймворк аутентификации вставляется в стандартные настройки проекта. Он состоит из приложения `django.contrib.auth` и следующих ниже двух классов промежуточных программных компонентов, которые находятся в настроечном параметре `MIDDLEWARE` проекта:

- `AuthenticationMiddleware`: ассоциирует пользователей с запросами с помощью сеансов;
- `SessionMiddleware`: оперирует текущим сеансом во всех запросах.

Промежуточные программные компоненты состоят из классов с методами, которые исполняются глобально в фазе запроса или ответа. Классы промежуточных программных компонентов будут использоваться несколько раз на протяжении этой книги, а о том, как создавать конкретно-прикладной промежуточный программный компонент, вы узнаете в главе 17 «Выход в прямой эфир».

Фреймворк аутентификации также содержит следующие ниже модели, которые определены в `django.contrib.auth.models`:

- `User`: модель пользователя с базовыми полями; главные поля этой модели таковы: `username`, `password`, `email`, `first_name`, `last_name` и `is_active`;
- `Group`: модель группы для отнесения пользователей к категориям;
- `Permission`: флаги для пользователей или групп для выполнения ими определенных действий.

Фреймворк аутентификации также содержит стандартные представления и формы для аутентификации, которые будут использоваться позже.

Создание представления входа в систему

Мы начнем этот раздел с использования встроенного в Django фреймворка аутентификации с целью предоставления пользователям возможности входить в систему. Мы создадим представление, которое будет выполнять следующие ниже действия по входу пользователя в систему:

- показывать пользователю форму для входа в систему;
- получать пользовательское имя и пароль, предоставляемые пользователем при передаче формы на обработку;
- аутентифицировать пользователя по данным, хранящимся в базе данных;
- проверять, что пользователь активен;
- регистрировать вход пользователя в систему и начинать сеанс аутентификации.

Мы начнем с компоновки формы для входа в систему.

Внутри каталога приложения `account` создайте новый файл `forms.py` и добавьте в него следующие ниже строки:

```
from django import forms

class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)
```

Приведенная выше форма будет использоваться для аутентификации пользователей по базе данных. Обратите внимание, что для прорисовки HTML-элемента `password` используется виджет `PasswordInput`. Такой подход позволит вставлять `type="password"` в HTML, чтобы браузер воспринимал его как ввод пароля.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код:

```
from django.http import HttpResponse
from django.shortcuts import render
from django.contrib.auth import authenticate, login
from .forms import LoginForm

def user_login(request):
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            user = authenticate(request,
                                username=cd['username'],
                                password=cd['password'])
            if user is not None:
```

```
                if user.is_active:
                    login(request, user)
                    return HttpResponse('Authenticated successfully')
                else:
                    return HttpResponse('Disabled account')
            else:
                return HttpResponse('Invalid login')
    else:
        form = LoginForm()
    return render(request, 'account/login.html', {'form': form})
```

Базовое представление входа в систему делает следующее.

При вызове представления `user_login` с запросом методом `GET` посредством инструкции `form = LoginForm()` создается экземпляр новой формы входа. Затем эта форма передается в шаблон.

Когда пользователь передает форму методом `POST`, выполняются следующие ниже действия:

- посредством инструкции `form = LoginForm(request.POST)` создается экземпляр формы с переданными данными;
- форма валидируется методом `form.is_valid()`. Если она невалидна, то ошибки формы будут выведены позже в шаблоне (например, если пользователь не заполнил одно из полей);
- если переданные на обработку данные валидны, то пользователь аутентифицируется по базе данных методом `authenticate()`. Указанный метод принимает объект `request`, параметры `username` и `password` и возвращает объект `User`, если пользователь был успешно аутентифицирован, либо `None` в противном случае. Если пользователь не был успешно аутентифицирован, то возвращается сырой ответ `HttpResponse` с сообщением **Invalid login** (Недопустимый логин);
- если пользователь успешно аутентифицирован, то статус пользователя проверяется путем обращения к атрибуту `is_active`. Указанный атрибут принадлежит модели `User` веб-фреймворка Django. Если пользователь не активен, то возвращается `HttpResponse` с сообщением **Disabled account** (Отключенная учетная запись);
- если пользователь активен, то он входит в систему. Пользователь задается в сеансе путем вызова метода `login()`. При этом возвращается сообщение **Authenticated successfully** (Аутентификация прошла успешно).

Обратите внимание на разницу между `authenticate()` и `login()`: `authenticate()` проверяет учетные данные пользователя и возвращает объект `User`, если они правильны; `login()` задает пользователя в текущем сеансе.

Теперь мы создадим шаблон URL-адреса этого представления.

Внутри каталога приложения `account` создайте новый файл `urls.py` и добавьте в него следующий ниже исходный код:

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

urlpatterns = [
    path('login/', views.user_login, name='login'),
]
```

Отредактируйте расположенный в каталоге проекта `bookmarks` главный файл `urls.py`, импортировав `include` и добавив шаблоны URL-адресов приложения `account`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
from django.contrib import admin
from django.urls import path, include

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

Теперь появляется возможность обращаться к представлению входа в систему по URL-адресу.

Давайте создадим шаблон этого представления. Поскольку в проекте еще шаблонов нет, мы начнем с создания базового шаблона, который будет расширен шаблоном входа в систему.

Внутри каталога приложения `account` создайте следующие ниже файлы и каталоги:

```
templates/
    account/
        login.html
    base.html
```

Отредактируйте шаблон `base.html`, добавив следующий ниже исходный код:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
  <div id="header">
    <span class="logo">Bookmarks</span>
```

```
    </div>
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
  </body>
  </html>
```

Это будет базовый шаблон веб-сайта. Как и в предыдущем проекте, вставьте в базовый шаблон стили CSS. Статические файлы CSS находятся в прилагаемом к данной главе исходном коде. Скопируйте каталог `static/` приложения `account` из исходного кода главы в то же место в вашем проекте, чтобы иметь возможность использовать статические файлы. Содержимое каталога находится на странице https://github.com/PacktPublishing/Django-4-by-Example/tree/master/Chapter04/bookmarks/account/static.

В базовом шаблоне определяется блок `title` и блок `content`. Расширяющие его шаблоны заполняют эти блоки контентом.

Давайте заполним шаблон формы входа в систему.

Откройте шаблон `account/login.html` и добавьте в него следующий ниже исходный код:

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

{% block title %}Log-in{% endblock %}

{% block content %}
  <h1>Log-in</h1>
  <p>Please, use the following form to log-in:</p>
  <form method="post">
    {{ form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Log in"></p>
  </form>
```

Приведенный выше шаблон содержит форму, экземпляр которой создается в представлении. Поскольку эта форма будет передаваться на обработку методом `POST`, вставляется и шаблонный тег `{% csrf_token %}`, чтобы защититься от подделки межсайтовых запросов (CSRF). Вы узнали о защите от CSRF в главе 2 «Усовершенствование блога за счет продвинутых функциональностей».

В базе данных пока пользователей нет. Сначала необходимо создать суперпользователя, чтобы получить доступ к сайту администрирования и управлять другими пользователями.

Исполните следующую ниже команду в командной оболочке:

```
python manage.py createsuperuser
```

Вы увидите следующее ниже сообщение. Введите желаемое пользовательское имя, электронную почту и пароль, как показано ниже:

```
Username (leave blank to use 'admin'): admin
Email address: admin@admin.com
Password: ********
Password (again): ********
```

После этого вы увидите такое сообщение об успехе:

```
Superuser created successfully.
```

Следующей ниже командой запустите сервер разработки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/` в своем браузере. Войдите на сайт администрирования, используя учетные данные только что созданного вами пользователя. Вы увидите встроенный в Django сайт администрирования, включая модели `User` и `Group` фреймворка аутентификации.

Он будет выглядеть, как показано ниже:

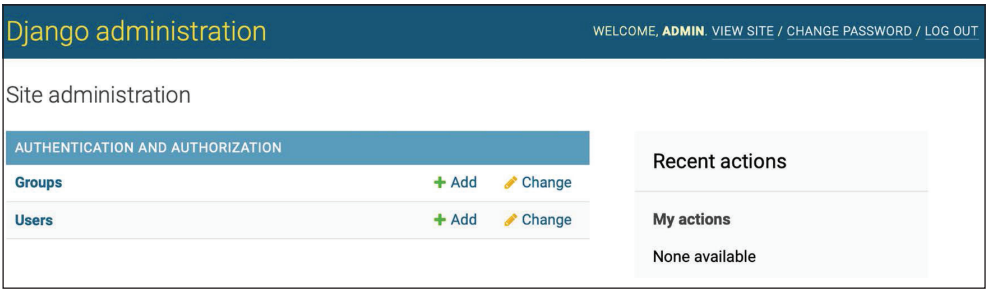

Рис. 4.1. Индексная страница встроенного в Django сайта администрирования, включая Users и Groups

В строке **Users** (Пользователи) кликните по ссылке **Add** (Добавить).

Создайте нового пользователя с помощью сайта администрирования, как показано ниже:

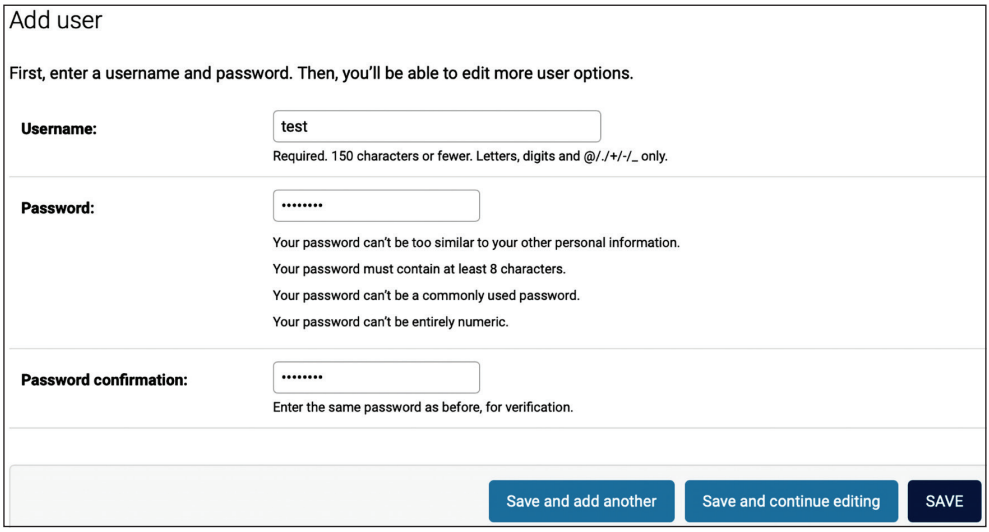

Рис. 4.2. Форма Add user (Добавить пользователя) на сайте администрирования

Введите данные пользователя и кликните по кнопке **SAVE** (Сохранить), чтобы сохранить нового пользователя в базе данных.

Затем в разделе **Personal info** (Личная информация) заполните поля **First name** (Имя), **Last name** (Фамилия) и **Email address** (Адрес электронной почты), как показано ниже, и кликните по кнопке **Save** (Сохранить), чтобы сохранить изменения:

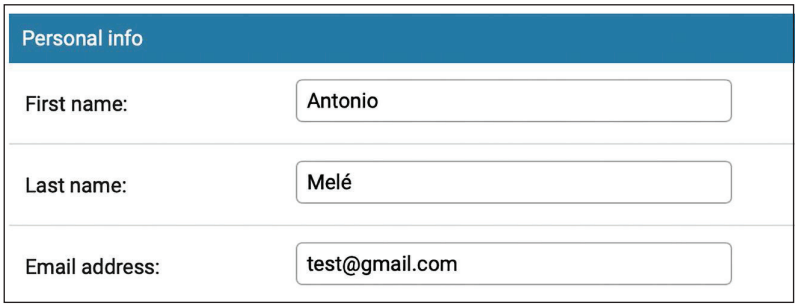

Рис. 4.3. Форма редактирования пользователя на сайте администрирования

Пройдите по URL-адресу `http://127.0.0.1:8000/account/login/` в своем браузере. Вы должны увидеть прорисованный шаблон, включая форму входа в систему:

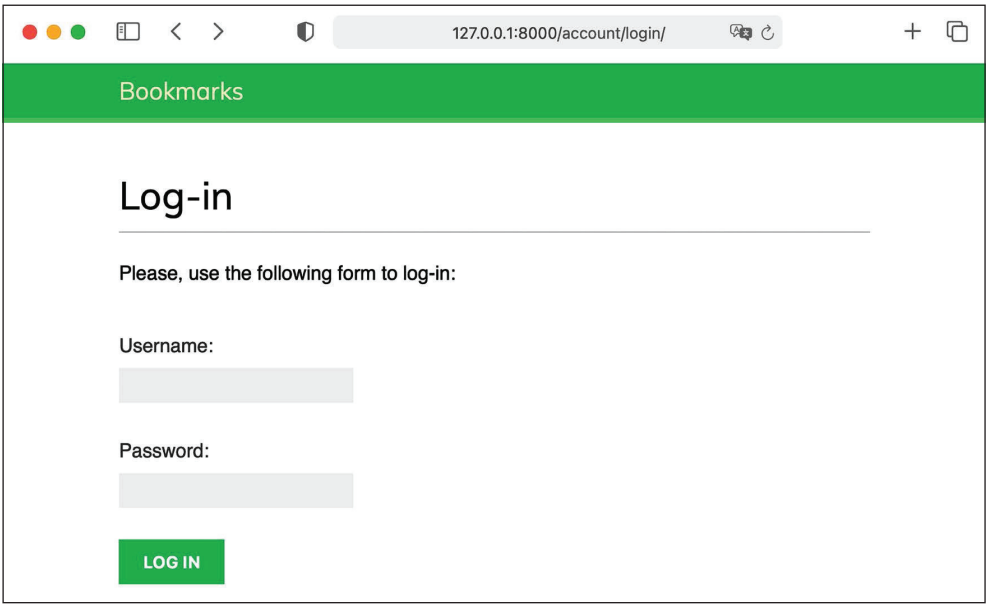

Рис. 4.4. Страница входа пользователя в систему

Введите недопустимые учетные данные и передайте форму на обработку. Вы должны получить следующий ответ **Invalid login** (Недопустимый логин):

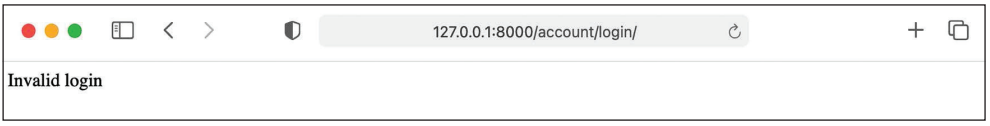

Рис. 4.5. Ответ в виде простого текста о недопустимом логине

Введите допустимые учетные данные; вы получите следующий ниже ответ **Authenticated successfully** (Аутентификация прошла успешно):

Рис. 4.6. Ответ в виде простого текста об успешной аутентификации

Вы научились аутентифицировать пользователей и создавать свое собственное представление аутентификации. Хотя можно разрабатывать свои собственные представления аутентификации, Django имеет готовые и служащие для этой цели представления, которые можно задействовать в своих проектах.

Использование встроенных в Django представлений аутентификации

Во встроенном Django фреймворке аутентификации содержится несколько форм и представлений, которые можно использовать сразу же. Созданное представление входа является хорошим упражнением на понимание процесса аутентификации пользователей в Django. Однако в большинстве случаев можно использовать стандартные представления аутентификации.

Django предоставляет следующие представления на основе классов для работы с аутентификацией. Все они расположены в `django.contrib.auth.views`:

- `LoginView`: оперирует формой входа и регистрирует вход пользователя;
- `LogoutView`: регистрирует выход пользователя.

Django предоставляет следующие ниже представления для оперирования сменой пароля:

- `PasswordChangeView`: оперирует формой для смены пароля пользователя;
- `PasswordChangeDoneView`: представление страницы об успехе, на которую пользователь перенаправляется после успешной смены пароля.

Django также содержит следующие ниже представления, позволяющие пользователям сбрасывать свой пароль:

- `PasswordResetView`: позволяет пользователям сбрасывать свой пароль. Генерирует одноразовую ссылку с токеном и отправляет ее на электронный ящик пользователя;
- `PasswordResetDoneView`: сообщает пользователям, что им было отправлено электронное письмо, содержащее ссылку на сброс пароля;
- `PasswordResetConfirmView`: позволяет пользователям устанавливать новый пароль;
- `PasswordResetCompleteView`: представление страницы об успехе, на которую пользователь перенаправляется после успешного сброса пароля.

Описанные выше представления сэкономят время при разработке любого веб-приложения с использованием учетных записей пользователей. В указанных представлениях используются стандартные значения, которые можно переопределять, например расположение прорисовываемого шаблона или используемая в представлении форма.

Более подробную информацию о встроенных представлениях аутентификации можно получить по адресу https://docs.djangoproject.com/en/4.1/topics/auth/default/#all-authentication-views.

Представления входа и выхода

Отредактируйте файл `urls.py` приложения `account`, добавив исходный код, выделенный жирным шрифтом:

```
from django.urls import path
from django.contrib.auth import views as auth_views
from . import views

urlpatterns = [
    # предыдущий url входа
    # path('login/', views.user_login, name='login'),

    # url-адреса входа и выхода
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

В приведенном выше исходном коде закомментирован шаблон URL-адреса созданного ранее представления `user_login`. Теперь будет использоваться представление `LoginView` из фреймворка аутентификации. Также был добавлен шаблон URL-адреса представления `LogoutView`.

Создайте новый каталог внутри каталога `templates/` приложения `account` и назовите его `registration`. Это стандартный путь, по которому представления аутентификации будут обращаться к шаблонам аутентификации, если не указан иной.

Модуль `django.contrib.admin` вставляет используемые на сайте администрирования шаблоны аутентификации, например шаблон входа. Поместив приложение `account` в начало настроечного параметра `INSTALLED_APPS` при конфигурировании проекта, мы обеспечили, чтобы вместо шаблонов аутентификации, определенных в любом другом приложении, Django использовал наши шаблоны аутентификации.

Внутри каталога `templates/registration/` создайте новый файл, назовите его `login.html` и добавьте в него следующий ниже исходный код:

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

{% block title %}Log-in{% endblock %}

{% block content %}
  <h1>Log-in</h1>
  {% if form.errors %}
    <p>
      Your username and password didn't match.
      Please try again.
    </p>
  {% else %}
    <p>Please, use the following form to log-in:</p>
  {% endif %}
  <div class="login-form">
    <form action="{% url 'login' %}" method="post">
      {{ form.as_p }}
```

```
      {% csrf_token %}
      <input type="hidden" name="next" value="{{ next }}" />
      <p><input type="submit" value="Log-in"></p>
    </form>
  </div>
{% endblock %}
```

Этот шаблон входа очень похож на созданный ранее. По умолчанию в Django используется форма `AuthenticationForm`, расположенная в `django.contrib.auth.forms`. Эта форма пытается аутентифицировать пользователя и выдает ошибку валидации, если вход оказался безуспешным. В шаблоне используется тег `{% if form.errors %}`, чтобы выполнять проверку на ошибочность предоставленных учетных данных.

Мы добавили скрытый HTML-элемент `<input>`, чтобы передавать на обработку значение переменной `next`. Если передать параметр с именем `next` в запрос, например обращаясь по `http://127.0.0.1:8000/account/login/?next=/account/`, то эта переменная передается представлению входа.

Следующим параметром должен быть URL-адрес. Если этот параметр задан, то встроенное в Django представление входа будет перенаправлять пользователя на указанный URL-адрес после успешного входа.

Теперь внутри каталога `templates/registration/` создайте шаблон `logged_out.html` и придайте ему следующий вид:

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

{% block title %}Logged out{% endblock %}

{% block content %}
  <h1>Logged out</h1>
  <p>
    You have been successfully logged out.
    You can <a href="{% url "login" %}">log-in again</a>.
  </p>
{% endblock %}
```

Это шаблон, который будет отображаться после выхода пользователя.

Мы добавили шаблоны URL-адресов и шаблоны представлений входа и выхода. Теперь пользователи могут входить и выходить, используя встроенные в Django представления аутентификации.

Сейчас мы создадим новое представление с целью отображения информационной панели при входе пользователей в свои учетные записи.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код:

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

```
@login_required
def dashboard(request):
    return render(request,
                  'account/dashboard.html',
                  {'section': 'dashboard'})
```

Мы создали представление `dashboard` и применили к нему декоратор `login_required` из фреймворка аутентификации. Декоратор `login_required` проверяет аутентификацию текущего пользователя.

Если пользователь аутентифицирован, то оно исполняет декорированное представление; если пользователь не аутентифицирован, то оно перенаправляет пользователя на URL-адрес входа с изначально запрошенным URL-адресом в качестве `GET`-параметра с именем `next`.

При таком подходе представление входа перенаправляет пользователей на URL-адрес, к которому они пытались обратиться после успешного входа. Напомним, что с этой целью в шаблон входа был добавлен скрытый HTML-элемент `<input>` с именем `next`.

Мы также определили переменную `section`. Эта переменная будет использоваться для подсвечивания текущего раздела в главном меню сайта.

Далее необходимо создать шаблон представления `dashboard`.

Внутри каталога `templates/account/` создайте новый файл и назовите его `dashboard.html`. Добавьте в него следующий ниже исходный код:

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

{% block title %}Dashboard{% endblock %}

{% block content %}
  <h1>Dashboard</h1>
  <p>Welcome to your dashboard.</p>
{% endblock %}
```

Отредактируйте файл `urls.py` приложения `account`, добавив следующий ниже шаблон URL-адреса представления. Новый исходный код выделен жирным шрифтом:

```
urlpatterns = [
    # предыдущий url-адрес входа
    # path('login/', views.user_login, name='login'),

    # url-адреса входа и выхода
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    path('', views.dashboard, name='dashboard'),
]
```

Отредактируйте файл `settings.py` проекта, добавив следующий ниже исходный код:

```
LOGIN_REDIRECT_URL = 'dashboard'
LOGIN_URL = 'login'
LOGOUT_URL = 'logout'
```

Мы определили следующие настроечные параметры:

- `LOGIN_REDIRECT_URL`: сообщает Django URL-адрес, на который следует перенаправлять пользователя после успешного входа, если в запросе нет параметра `next`;
- `LOGIN_URL`: URL-адрес, на который следует перенаправлять пользователя, чтобы зарегистрировать его вход (например, представления, в которых используется декоратор `login_required`);
- `LOGOUT_URL`: URL-адрес, на который следует перенаправлять пользователя, чтобы зарегистрировать его выход.

В шаблонах URL-адресов были использованы имена URL-адресов, которые были определены ранее с помощью атрибута `name` функции `path()`. Вместо имен URL-адресов в этих настроечных параметрах также можно использовать жестко привязанные URL-адреса.

Давайте подведем итоги того, что мы сделали к этому моменту:

- в проект были добавлены встроенные во фреймворк аутентификации представления входа и выхода;
- для обоих представлений были созданы конкретно-прикладные шаблоны и определено простое представление информационной панели, куда перенаправлять пользователей после их входа;
- наконец, добавлены настроечные параметры, чтобы Django использовал эти URL-адреса по умолчанию.

Теперь надо добавить ссылки на страницы входа и выхода в базовый шаблон. Для этого необходимо выяснить, вошел ли текущий пользователь в систему или нет, чтобы отображать надлежащую ссылку, соответствующую каждому случаю. Текущий пользователь задается в объекте `HttpRequest` промежуточным компонентом аутентификации. К нему можно обращаться через `request.user`. Объект `User` находится в запросе, даже если пользователь не аутентифицирован. Не прошедший аутентификацию пользователь задается в запросе как экземпляр `AnonymousUser`. Самый лучший способ проверить аутентификацию текущего пользователя – обратиться к доступному только для чтения атрибуту `is_authenticated`.

Отредактируйте шаблон `templates/base.html`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
```

```
    <link href="{% static "css/base.css" %}" rel="stylesheet">
  </head>
  <body>
    <div id="header">
      <span class="logo">Bookmarks</span>
      {% if request.user.is_authenticated %}
        <ul class="menu">
          <li {% if section == "dashboard" %}class="selected"{% endif %}>
            <a href="{% url "dashboard" %}">My dashboard</a>
          </li>
          <li {% if section == "images" %}class="selected"{% endif %}>
            <a href="#">Images</a>
          </li>
          <li {% if section == "people" %}class="selected"{% endif %}>
            <a href="#">People</a>
          </li>
        </ul>
      {% endif %}
      <span class="user">
        {% if request.user.is_authenticated %}
          Hello {{ request.user.first_name|default:request.user.username }},
          <a href="{% url "logout" %}">Logout</a>
        {% else %}
          <a href="{% url "login" %}">Log-in</a>
        {% endif %}
      </span>
    </div>
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
  </body>
  </html>
```

Меню сайта отображается только для аутентифицированных пользователей. При этом проверяется переменная `section`, чтобы добавить атрибут `selected` CSS-класса в списковый пункт `<li>` меню; указанный пункт меню относится к текущему разделу. Благодаря этому соответствующий текущему разделу пункт меню будет выделяться с помощью CSS. Если пользователь аутентифицирован, то отображается настоящее имя пользователя и ссылка на страницу выхода; в противном случае отображается ссылка на страницу входа. Если настоящее имя пользователя является пустым, то вместо него отображается пользовательское имя (username) при помощи `request.user.first_name|default:request.user.username`.

Пройдите по URL-адресу `http://127.0.0.1:8000/account/login/` в своем браузере. Вы должны увидеть страницу входа. Введите допустимое пользовательское имя и пароль и кликните по кнопке **Log-in** (Войти). Вы должны увидеть следующий ниже экран:

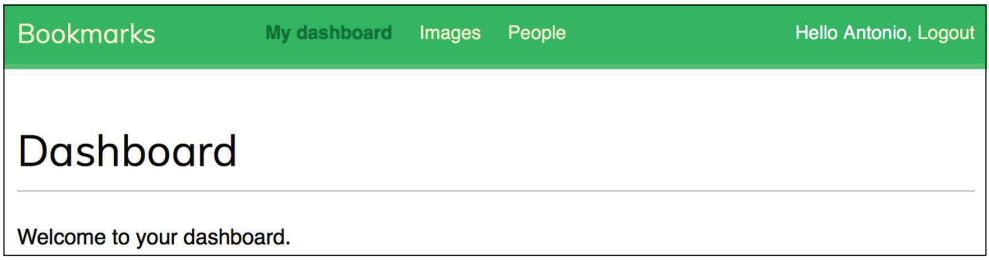

Рис. 4.7. Страница информационной панели

Пункт меню **My dashboard** (Моя информационная панель) выделен с помощью CSS, так как имеет класс `selected`. Поскольку пользователь аутентифицирован, его имя отображается в правой части заголовка. Кликните по ссылке **Logout** (Выйти). Вы должны увидеть следующую ниже страницу:

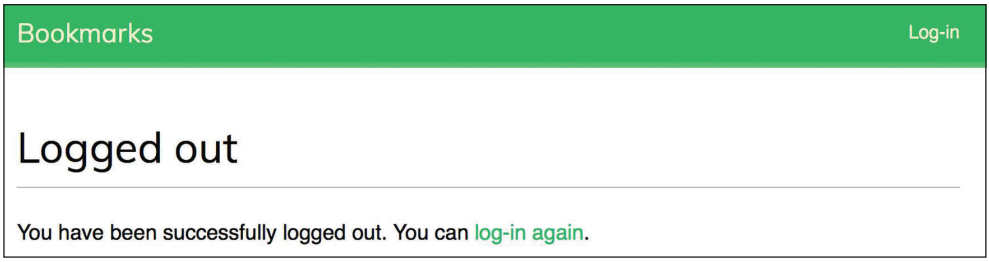

Рис. 4.8. Страница информации о выходе из системы

На этой странице видно, что пользователь вышел из системы, и, следовательно, меню сайта не отображается. Теперь отображаемая в правой части заголовка ссылка называется **Log-in** (Войти).

Если вместо своей собственной страницы Logged out (Зарегистрирован выход) вы видите страницу Logged out сайта администрирования, то следует обратиться к настроечному параметру `INSTALLED_APPS` проекта и проверить, чтобы приложение `django.contrib.admin` располагалось после приложения `account`. Оба приложения содержат шаблоны выхода, которые расположены по одному и тому же относительному пути. Загрузчик шаблонов Django будет прокручивать различные приложения в списке `INSTALLED_APPS` и использовать первый найденный шаблон.

Представления смены пароля

Далее необходимо обеспечить пользователям возможность менять свой пароль после входа в систему. Мы выполним интеграцию встроенных в Django представлений аутентификации, предназначенных для смены пароля.

Откройте файл `urls.py` приложения `account` и добавьте следующие ниже шаблоны URL-адресов, выделенные жирным шрифтом:

```
urlpatterns = [
    # предыдущий url-адрес входа
    # path('login/', views.user_login, name='login'),
    # url-адреса входа и выхода
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # url-адреса смены пароля
    path('password-change/',
         auth_views.PasswordChangeView.as_view(),
         name='password_change'),
    path('password-change/done/',
         auth_views.PasswordChangeDoneView.as_view(),
         name='password_change_done'),

    path('', views.dashboard, name='dashboard'),
]
```

Представление `PasswordChangeView` будет работать с формой для смены пароля, а представление `PasswordChangeDoneView` будет отображать сообщение об успехе, после того как пользователь успешно сменит свой пароль. Давайте создадим шаблон каждого представления.

Добавьте новый файл в каталог `templates/registration/` приложения `account` и назовите его `password_change_form.html`. Добавьте в него следующий ниже исходный код:

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

{% block title %}Change your password{% endblock %}

{% block content %}
  <h1>Change your password</h1>
  <p>Use the form below to change your password.</p>
  <form method="post">
    {{ form.as_p }}
    <p><input type="submit" value="Change"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

Шаблон `password_change_form.html` содержит форму для смены пароля.

Теперь в том же каталоге создайте еще один файл и назовите его `password_change_done.html`. Добавьте в него следующий ниже исходный код:

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

{% block title %}Password changed{% endblock %}

{% block content %}
  <h1>Password changed</h1>
  <p>Your password has been successfully changed.</p>
{% endblock %}
```

Шаблон `password_change_done.html` содержит только сообщение об успехе, которое будет отображаться, когда пользователь успешно сменит свой пароль.

Пройдите по URL-адресу `http://127.0.0.1:8000/account/password-change/` в своем браузере. Если вы не вошли, то браузер перенаправит на страницу **Log-in** (Войти). После успешной аутентификации вы увидите следующую ниже страницу смены пароля:

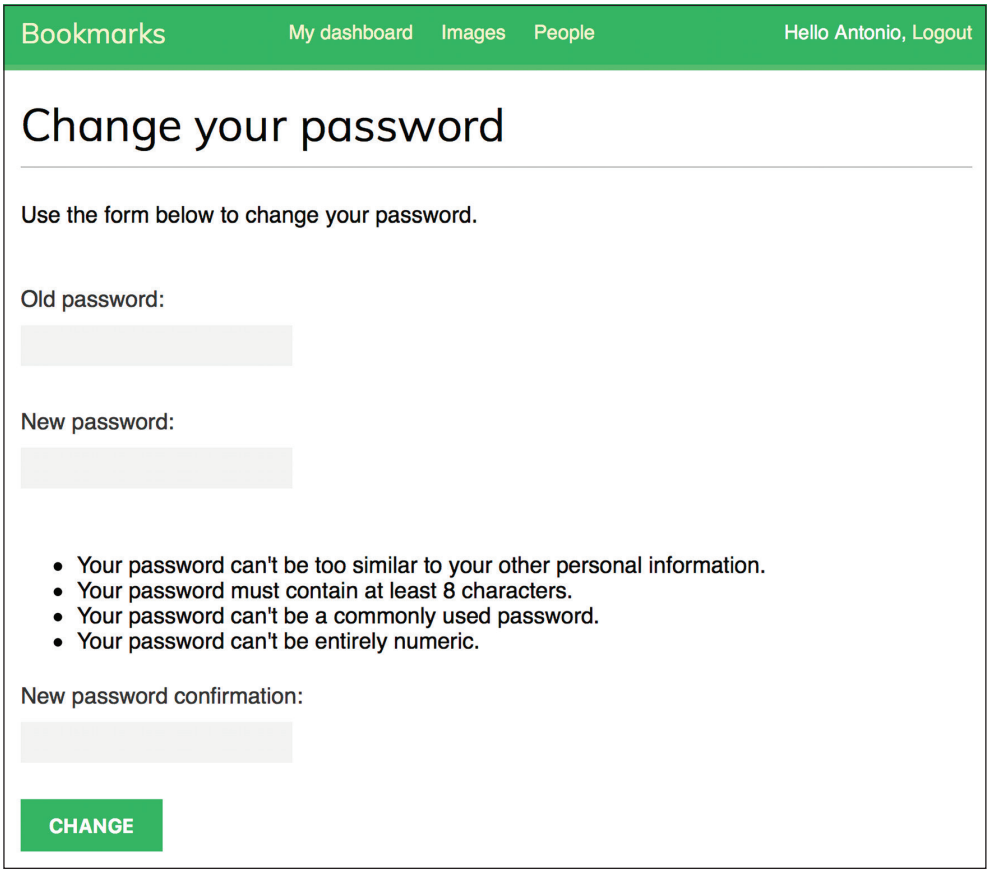

Рис. 4.9. Форма для смены пароля

Заполните форму текущим паролем и новым паролем и кликните по кнопке **CHANGE** (Сменить).

Вы увидите следующую ниже страницу успешной смены пароля:

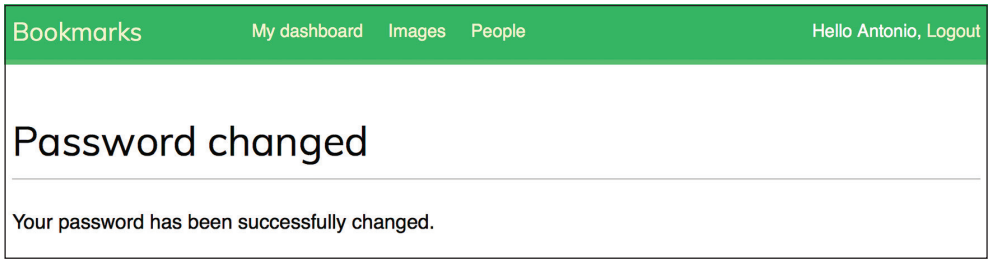

Рис. 4.10. Страница успешной смены пароля

Выйдите и снова войдите, используя новый пароль, чтобы убедиться, что все работает так, как и ожидалось.

Представление сброса пароля

Отредактируйте файл `urls.py` приложения `account`, добавив следующие ниже шаблоны URL-адресов, выделенные жирным шрифтом:

```
urlpatterns = [
    # path('login/', views.user_login, name='login'),

    # url-адреса входа и выхода
    path('login/', auth_views.LoginView.as_view(), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # url-адреса смены пароля
    path('password-change/',
         auth_views.PasswordChangeView.as_view(),
         name='password_change'),
    path('password-change/done/',
          auth_views.PasswordChangeDoneView.as_view(),
          name='password_change_done'),

    # url-адреса сброса пароля
    path('password-reset/',
         auth_views.PasswordResetView.as_view(),
         name='password_reset'),
    path('password-reset/done/',
         auth_views.PasswordResetDoneView.as_view(),
         name='password_reset_done'),
```

```
    path('password-reset/<uidb64>/<token>/',
         auth_views.PasswordResetConfirmView.as_view(),
         name='password_reset_confirm'),
    path('password-reset/complete/',
         auth_views.PasswordResetCompleteView.as_view(),
         name='password_reset_complete'),

    path('', views.dashboard, name='dashboard'),
]
```

Добавьте новый файл в каталог `templates/registration/` приложения `account` и назовите его `password_reset_form.html`. Добавьте в него следующий ниже исходный код:

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

{% block title %}Reset your password{% endblock %}

{% block content %}
  <h1>Forgotten your password?</h1>
  <p>Enter your e-mail address to obtain a new password.</p>
  <form method="post">
    {{ form.as_p }}
    <p><input type="submit" value="Send e-mail"></p>
    {% csrf_token %}
  </form>
{% endblock %}
```

Теперь в том же каталоге создайте еще один файл и назовите его `password_reset_email.html`. Добавьте в него следующий ниже исходный код[1]:

```
Someone asked for password reset for email {{ email }}.
Follow the link below: {{ protocol }}://{{ domain }}{% url "password_reset_confirm"
uidb64=uid token=token %}
Your username, in case you've forgotten: {{ user.get_username }}
```

Шаблон `password_reset_email.html` будет использоваться для отображения отправляемого пользователям электронного письма, чтобы сбросить свой пароль. Оно содержит токен сброса, который генерируется представлением.

В том же каталоге создайте еще один файл и назовите его `password_reset_done.html`. Добавьте в него следующий ниже исходный код:

---

[1] Кто-то сделал запрос на сброс пароля электронной почты {}. Пройдите по ссылке: {}. Ваше пользовательское имя, если вы забыли: {}. – *Прим. перев.*

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

{% block title %}Reset your password{% endblock %}

{% block content %}
  <h1>Reset your password</h1>
  <p>We've emailed you instructions for setting your password.</p>
  <p>If you don't receive an email, please make sure
     you've entered the address you registered with.</p>
{% endblock %}
```

В том же каталоге создайте еще один шаблон и назовите его `password_reset_confirm.html`. Добавьте в него следующий ниже исходный код:

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

{% block title %}Reset your password{% endblock %}

{% block content %}
  <h1>Reset your password</h1>
  {% if validlink %}
    <p>Please enter your new password twice:</p>
    <form method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <p><input type="submit" value="Change my password" /></p>
    </form>
  {% else %}
    <p>The password reset link was invalid, possibly because it has already
been used. Please request a new password reset.</p>
  {% endif %}
{% endblock %}
```

В этом шаблоне мы подтверждаем валидность/невалидность ссылки на сброс пароля, проверяя переменную `validlink`. Представление `PasswordResetConfirmView` проверяет валидность указанного в URL-адресе токена и передает переменную `validlink` в шаблон. Если ссылка валидна, то отображается форма для сброса пароля пользователя. Пользователи могут устанавливать новый пароль, только если у них есть валидная ссылка на сброс пароля.

Создайте еще один шаблон и назовите его `password_reset_complete.html`. Введите в него следующий ниже исходный код:

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

{% block title %}Password reset{% endblock %}

{% block content %}
  <h1>Password set</h1>
  <p>Your password has been set.
     You can <a href="{% url "login" %}">log in now</a></p>
{% endblock %}
```

Наконец, отредактируйте шаблон `registration/login.html` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

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

{% block title %}Log-in{% endblock %}

{% block content %}
  <h1>Log-in</h1>
  {% if form.errors %}
    <p>
      Your username and password didn't match.
      Please try again.
    </p>
  {% else %}
    <p>Please, use the following form to log-in:</p>
  {% endif %}
  <div class="login-form">
    <form action="{% url 'login' %}" method="post">
      {{ form.as_p }}
      {% csrf_token %}
      <input type="hidden" name="next" value="{{ next }}" />
      <p><input type="submit" value="Log-in"></p>
    </form>
    <p>
      <a href="{% url "password_reset" %}">
        Forgotten your password?
      </a>
    </p>
  </div>
{% endblock %}
```

Теперь пройдите по URL-адресу `http://127.0.0.1:8000/account/login/` в своем браузере. На странице входа должна появиться ссылка на страницу сброса пароля, как показано ниже:

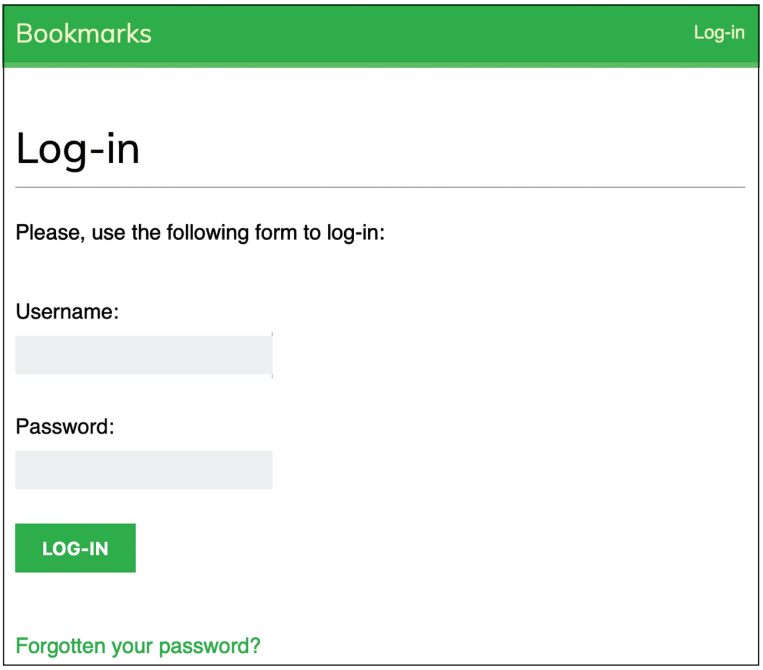

Рис. 4.11. Страница входа, включающая ссылку на страницу сброса пароля

Кликните по ссылке **Forgotten your password?** (Забыли свой пароль?). Вы увидите следующую ниже страницу:

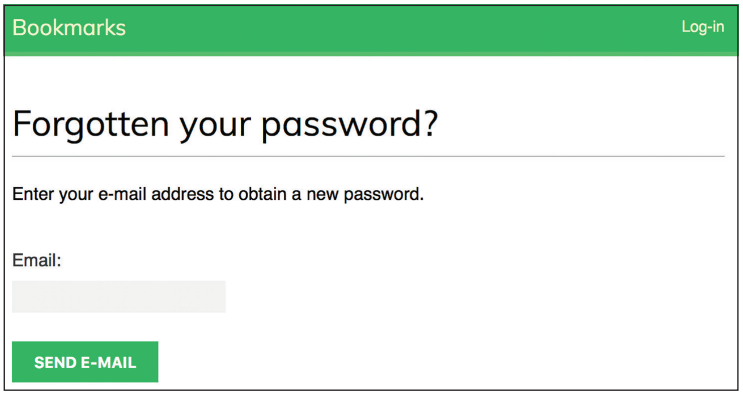

Рис. 4.12. Форма для восстановления пароля

На данном этапе нужно добавить конфигурацию простого протокола передачи почты (**SMTP**) в файл `settings.py` проекта, чтобы Django мог отправлять электронные письма. В главе 2 «Усовершенствование блога за счет продвинутых функциональностей» вы научились добавлять в проект настроечные па-

раметры электронной почты. Однако во время разработки имеется возможность конфигурировать Django таким образом, чтобы он писал электронные письма в стандартный вывод, а не отправлял их через SMTP-сервер. Django предоставляет почтовый бэкенд, позволяющий писать письма в консоль.

Отредактируйте файл `settings.py` проекта, добавив следующую ниже строку:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Настроечный параметр `EMAIL_BACKEND` указывает класс, который будет использоваться для отправки электронной почты.

Вернитесь в браузер, введите адрес электронной почты существующего пользователя и кликните по кнопке **SEND E-MAIL** (Отправить почту). Вы должны увидеть следующую ниже страницу:

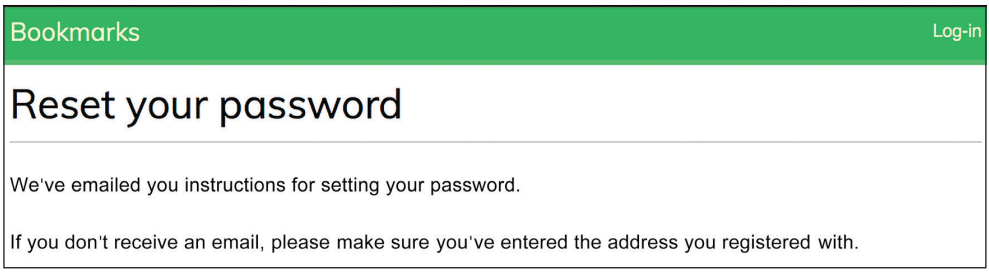

Рис. 4.13. Страница отправки сообщения электронной почты о сбросе пароля

Загляните в командную оболочку, в которой работает сервер разработки. Вы увидите сгенерированное электронное письмо, как показано ниже:

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Password reset on 127.0.0.1:8000
From: webmaster@localhost
To: test@gmail.com
Date: Mon, 10 Jan 2022 19:05:18 -0000
Message-ID: <162896791878.58862.14771487060402279558@MBP-amele.local>

Someone asked for password reset for email test@gmail.com. Follow the link below:
http://127.0.0.1:8000/account/password-reset/MQ/ardx0u-
b4973cfa2c70d652a190e79054bc479a/
Your username, in case you've forgotten: test
```

Письмо прорисовывается с помощью созданного ранее шаблона `password_reset_email.html`. URL-адрес сброса пароля содержит токен, который Django сгенерировал динамически.

Скопируйте URL-адрес из письма, который должен выглядеть примерно так: `http://127.0.0.1:8000/account/password-reset/MQ/ardx0u-b4973cfa2c70d652a190e79054bc479a/`, и откройте его в браузере. Вы должны увидеть следующую ниже страницу:

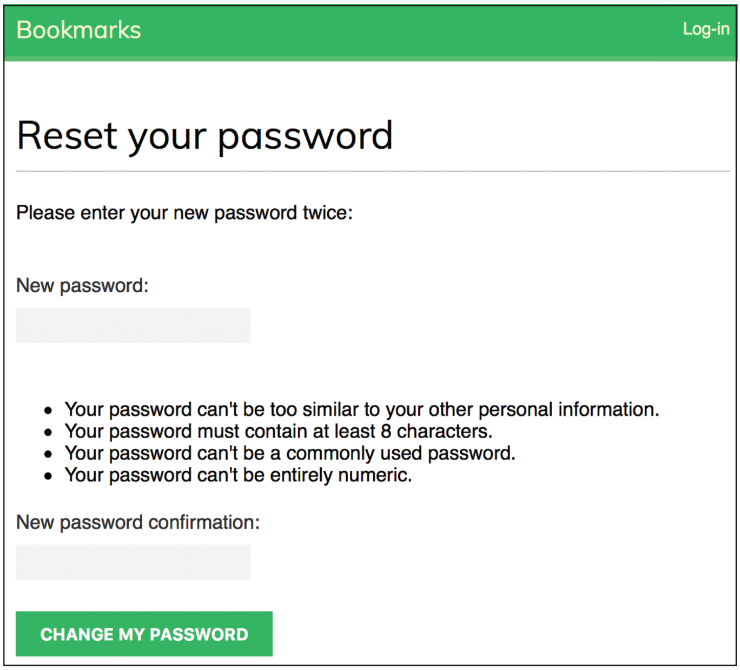

Рис. 4.14. Форма для сброса пароля

На странице установки нового пароля используется шаблон `password_reset_confirm.html`. Введите новое парольное слово и кликните по кнопке **CHANGE MY PASSWORD** (Сменить пароль). Django создаст новый хешированный пароль и сохранит его в базе данных. Вы увидите следующую ниже страницу с сообщением об успехе:

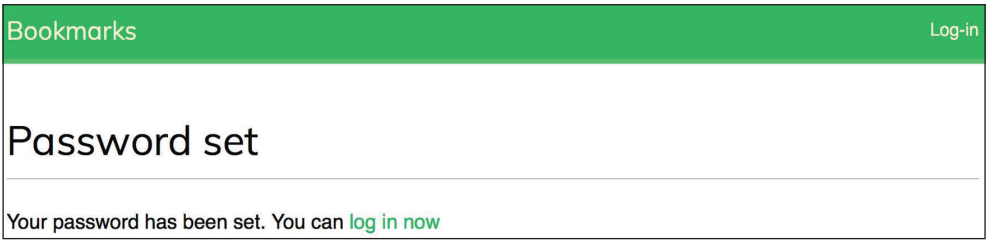

Рис. 4.15. Страница успешного сброса пароля

Теперь можно снова войти в учетную запись пользователя с новым паролем.

Каждый токен для установки нового пароля можно использовать только один раз. Если вы откроете полученную ссылку еще раз, то получите сообщение о том, что токен недействителен.

Мы только что интегрировали в проект представления, встроенные в фреймворк аутентификации. Эти представления подходят для большинства случаев. Однако если требуется другое поведение, то существует возможность создавать свои собственные представления.

Django предоставляет шаблоны URL-адресов представлений аутентификации, которые эквивалентны тем, которые мы только что создали. Мы заменим указанные шаблоны аутентификации на те, которые имеются в Django.

Закомментируйте шаблоны URL-адресов аутентификации, которые были добавлены в файл `urls.py` приложения `account`, и вместо них вставьте `django.contrib.auth.urls`, как показано ниже. Новый исходный код выделен жирным шрифтом:

```
from django.urls import path, include
from django.contrib.auth import views as auth_views
from . import views

urlpatterns = [
    # предыдущий url-адрес входа
    # path('login/', views.user_login, name='login'),

    # url-адреса входа и выхода
    # path('login/', auth_views.LoginView.as_view(), name='login'),
    # path('logout/', auth_views.LogoutView.as_view(), name='logout'),

    # url-адреса смены пароля
    # path('password-change/',
    #      auth_views.PasswordChangeView.as_view(),
    #      name='password_change'),
    # path('password-change/done/',
    #       auth_views.PasswordChangeDoneView.as_view(),
    #       name='password_change_done'),

    # url-адреса сброса пароля
    # path('password-reset/',
    #      auth_views.PasswordResetView.as_view(),
    #      name='password_reset'),
    # path('password-reset/done/',
    #      auth_views.PasswordResetDoneView.as_view(),
    #      name='password_reset_done'),
    # path('password-reset/<uidb64>/<token>/',
    #      auth_views.PasswordResetConfirmView.as_view(),
    #      name='password_reset_confirm'),
    # path('password-reset/complete/',
    #      auth_views.PasswordResetCompleteView.as_view(),
```

```
    #      name='password_reset_complete'),

    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
]
```

Встроенные шаблоны URL-адресов аутентификации можно посмотреть на странице https://github.com/django/django/blob/stable/4.0.x/django/contrib/auth/urls.py.

Теперь в проект были добавлены все необходимые представления аутентификации. Далее мы реализуем регистрацию пользователей.

Регистрация пользователей и профили пользователей

Теперь пользователи сайта могут входить в систему и выходить из системы, менять и сбрасывать пароль. Однако еще необходимо скомпоновать представление, позволяющее посетителям создавать учетную запись пользователя.

Регистрация пользователя

Давайте создадим простое представление регистрации пользователей в системе. Сначала нужно создать форму, чтобы пользователь мог вводить пользовательское имя, свое настоящее имя и пароль.

Отредактируйте файл `forms.py`, расположенный в каталоге приложения `account`, добавив в него следующие ниже строки, выделенные жирным шрифтом:

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

class LoginForm(forms.Form):
    username = forms.CharField()
    password = forms.CharField(widget=forms.PasswordInput)

class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password',
                               widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password',
                                widget=forms.PasswordInput)

    class Meta:
```

```
        model = User
        fields = ['username', 'first_name', 'email']
```

Здесь была создана модельная форма для модели пользователя. Данная форма содержит поля `username`, `first_name` и `email` модели `User`. Указанные поля будут валидироваться в соответствии с проверками на валидность соответствующих полей модели. Например, если пользователь выберет пользовательское имя, которое уже существует, то он получит ошибку валидации, поскольку пользовательское имя – это поле, определенное с использованием `unique=True`.

Далее были добавлены два дополнительных поля – `password` и `password2`, – для того чтобы пользователи могли устанавливать пароль и повторять его. Давайте добавим валидацию полей, чтобы проверить, что оба пароля одинаковы.

Отредактируйте файл `forms.py` в приложении учетной записи, добавив в класс `UserRegistrationForm` следующий ниже метод `clean_password2()`. Новый исходный код выделен жирным шрифтом:

```
class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password',
                               widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password',
                                widget=forms.PasswordInput)

    class Meta:
            model = User
            fields = ['username', 'first_name', 'email']

    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Passwords don\'t match.')
        return cd['password2']
```

Мы определили метод `clean_password2()`, чтобы сравнивать второй пароль с первым и выдавать ошибки валидации, если пароли не совпадают. Этот метод исполняется, когда форма проходит валидацию путем вызова ее метода `is_valid()`. Метод `clean_<fieladname>()` можно предоставлять любому полю формы, чтобы очищать значение или вызывать ошибку валидации формы для конкретного поля. Формы также содержат общий метод `clean()`, чтобы валидировать всю форму целиком, что бывает удобно при валидации полей, которые зависят друг от друга. В данном случае вместо переопределения метода `clean()` формы мы используем специфическую для поля валидацию `clean_password2()`. Такой подход позволяет избегать переопределения других специфических для полей проверок, которые `ModelForm` получает из ограничений, установленных в модели (например, валидация уникальности пользовательского имени `username`).

Django также предоставляет форму `UserCreationForm`, которая находится в `django.contrib.auth.forms` и очень похожа на созданную нами ранее.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.http import HttpResponse
from django.shortcuts import render
from django.contrib.auth import authenticate, login
from django.contrib.auth.decorators import login_required
from .forms import LoginForm, UserRegistrationForm

# ...

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Создать новый объект пользователя,
            # но пока не сохранять его
            new_user = user_form.save(commit=False)
            # Установить выбранный пароль
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Сохранить объект User
            new_user.save()
            return render(request,
                          'account/register_done.html',
                          {'new_user': new_user})
    else:
        user_form = UserRegistrationForm()
    return render(request,
                  'account/register.html',
                  {'user_form': user_form})
```

Представление создания учетных записей пользователей выглядит довольно просто. В целях безопасности вместо сохранения введенного пользователем необработанного пароля мы используем метод `set_password()` модели `User`. Данный метод хеширует пароль перед его сохранением в базе данных.

Django не хранит открытые текстовые пароли; вместо этого он хранит хешированные пароли. Хеширование – это процесс преобразования заданного ключа в другое значение. Хеш-функция используется для генерирования значения фиксированной длины в соответствии с математическим алгоритмом. Благодаря хешированию паролей с помощью безопасных алгорит-

мов Django гарантирует, что для взлома хранящихся в базе данных паролей пользователей потребуется огромное количество вычислительного времени.

Для хранения всех паролей по умолчанию используется алгоритм хеширования `PBKDF2` с хешем SHA256. Однако Django не только поддерживает проверку существующих паролей, хешированных с помощью `PBKDF2`, но и проверку сохраненных паролей, хешированных другими алгоритмами, такими как `PBKDF2SHA1`, `argon2`, `bcrypt` и `scrypt`.

Настроечный параметр `PASSWORD_HASHERS` определяет хешеры паролей, поддерживаемых проектом Django. Ниже приведен стандартный список хешеров паролей `PASSWORD_HASHERS`:

```
PASSWORD_HASHERS = [
    'django.contrib.auth.hashers.PBKDF2PasswordHasher',
    'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
    'django.contrib.auth.hashers.Argon2PasswordHasher',
    'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
    'django.contrib.auth.hashers.ScryptPasswordHasher',
]
```

Для хеширования всех паролей Django применяет первый элемент списка, в данном случае `PBKDF2PasswordHasher`. Остальные хешеры используются для проверки существующих паролей.

В Django 4.0 был введен хешер `scrypt`. Он более безопасен и рекомендуется чаще по сравнению с `PBKDF2`. Однако хешер `PBKDF2` все еще используется по умолчанию, так как `scrypt` требует наличия криптографической библиотеки OpenSSL 1.1+ и больше памяти.

Подробнее о том, как Django хранит пароли, и о задействованных хешерах паролей можно узнать на странице https://docs.djangoproject.com/en/4.1/topics/auth/passwords/.

Теперь отредактируйте файл `urls.py` приложения `account`, добавив следующий ниже шаблон URL-адреса, выделенный жирным шрифтом:

```
urlpatterns = [

    # ...

    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
]
```

Наконец, внутри каталога templates/account/template приложения account создайте новый шаблон, назовите его register.html и придайте ему следующий вид:

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

{% block title %}Create an account{% endblock %}

{% block content %}
  <h1>Create an account</h1>
  <p>Please, sign up using the following form:</p>
  <form method="post">
    {{ user_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Create my account"></p>
  </form>
{% endblock %}
```

В том же каталоге создайте дополнительный файл шаблона и назовите его register_done.html. Добавьте в него следующий ниже исходный код:

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

{% block title %}Welcome{% endblock %}

{% block content %}
  <h1>Welcome {{ new_user.first_name }}!</h1>
  <p>
    Your account has been successfully created.
    Now you can <a href="{% url "login" %}">log in</a>.
  </p>
{% endblock %}
```

Пройдите по URL-адресу http://127.0.0.1:8000/account/register/ в своем браузере. Вы увидите созданную вами страницу регистрации:

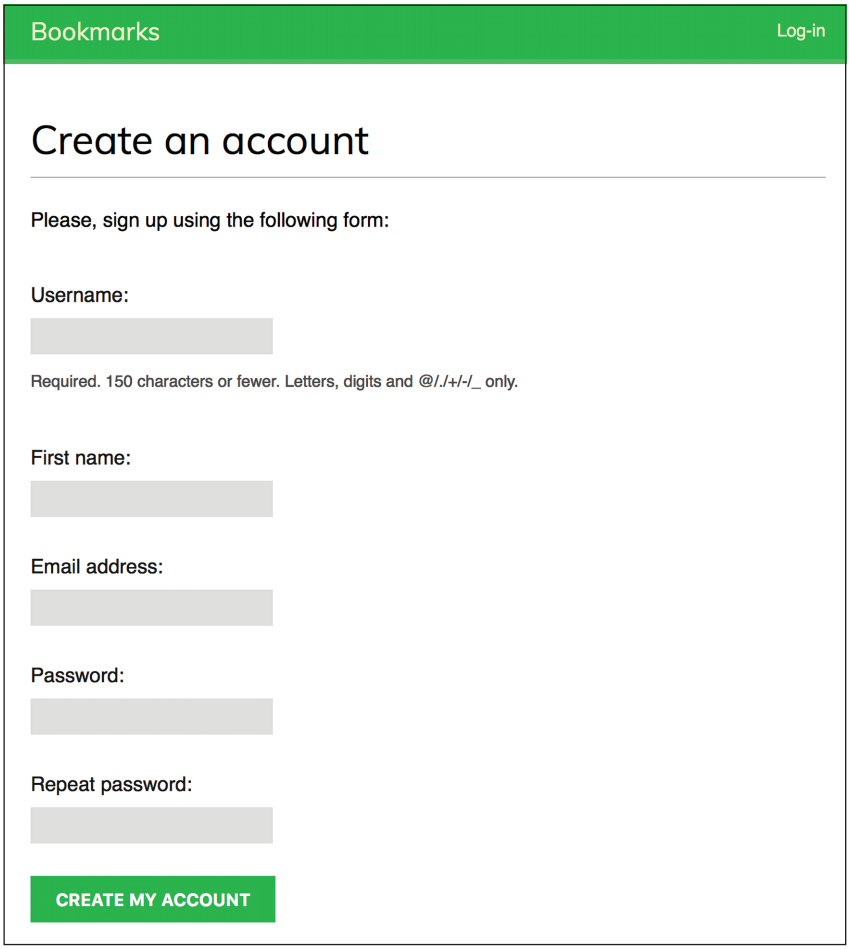

Рис. 4.16. Форма для создания учетной записи

Заполните данные нового пользователя и кликните по кнопке **CREATE MY ACCOUNT** (Создать учетную запись).

Если все поля валидны, то пользователь будет создан, и вы увидите следующее ниже сообщение об успехе:

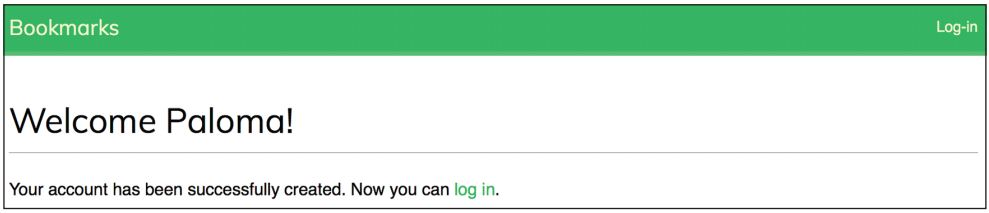

Рис. 4.17. Страница успешного создания учетной записи

Кликните по ссылке входа и введите свое пользовательское имя и пароль, чтобы убедиться, что вы можете получить доступ к только что созданной учетной записи.

Давайте добавим ссылку на регистрацию в шаблон входа. Откройте шаблон `registration/login.html` и найдите следующую ниже строку:

```
<p>Please, use the following form to log-in:</p>
```

Замените ее такими строками:

```
<p>
  Please, use the following form to log-in.
  If you don't have an account
  <a href="{% url "register" %}">register here</a>.
</p>
```

Пройдите по URL-адресу `http://127.0.0.1:8000/account/login/` в своем браузере. Теперь страница должна выглядеть, как показано ниже:

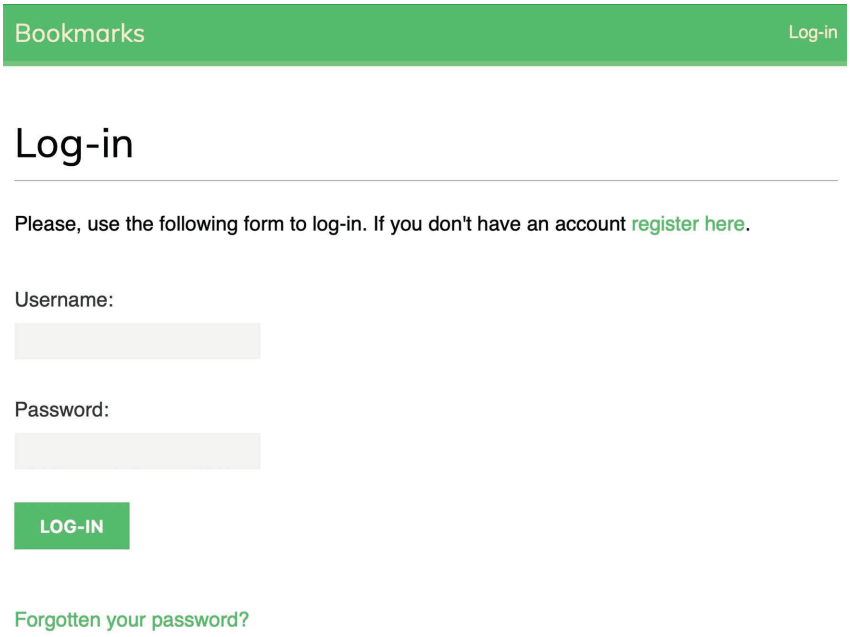

Рис. 4.18. Страница входа, в том числе ссылка на регистрацию

Мы сделали страницу регистрации доступной со страницы входа.

Расширение модели пользователя

Во время работы с учетными записями пользователей вы обнаружите, что модель `User` фреймворка аутентификации подходит для большинства распространенных случаев. Однако стандартная модель `User` идет в комплекте с ограниченным набором полей. Возможно, вы захотите расширить ее дополнительной информацией, которая будет релевантна для вашего приложения.

Простым способом расширения модели `User` является создание модели профиля, которая содержит взаимосвязи один-к-одному со встроенной в Django моделью `User` и любые дополнительные поля. Взаимосвязь один-к-одному похожа на поле `ForeignKey` с параметром `unique=True`. Обратной стороной взаимосвязи является неявная взаимосвязь один-к-одному со связанной моделью вместо менеджера для нескольких элементов. С каждой стороны взаимосвязи имеется доступ к одному связанному объекту.

Отредактируйте файл `models.py` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.db import models
from django.conf import settings

class Profile(models.Model):
    user = models.OneToOneField(settings.AUTH_USER_MODEL,
                                on_delete=models.CASCADE)
    date_of_birth = models.DateField(blank=True, null=True)
    photo = models.ImageField(upload_to='users/%Y/%m/%d/',
                              blank=True)

    def __str__(self):
        return f'Profile of {self.user.username}'
```

В целях поддержания обобщенного характера исходного кода следует использовать метод `get_user_model()`, который позволяет извлекать модель пользователя и настроечный параметр `AUTH_USER_MODEL`, чтобы ссылаться на него при определении связи модели с моделью пользователя, не ссылаясь на модель пользователя `auth` напрямую. Более подробную информацию об этом можно почитать по адресу https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#django.contrib.auth.get_user_model.

Профиль пользователя будет содержать дату рождения пользователя и его фотографию.

Поле `user` со взаимосвязью один-к-одному будет использоваться для ассоциирования профилей с пользователями. С помощью параметра `on_delete=models.CASCADE` мы принудительно удаляем связанный объект `Profile` при удалении объекта `User`.

Поле `date_of_birth` является экземпляром класса `DateField`. Мы сделали это поле опциональным посредством `blank=True`, и мы разрешаем нулевые значения посредством `null=True`.

Поле `photo` является экземпляром класса `ImageField`. Мы сделали это поле опциональным посредством `blank=True`. Класс `ImageField` управляет хранением файлов изображений. Он проверяет, что предоставленный файл является валидным изображением, сохраняет файл изображения в каталоге, указанном параметром `upload_to`, и сохраняет относительный путь к файлу в связанном поле базы данных. Класс `ImageField` по умолчанию транслируется в столбец `VARHAR(100)` в базе данных. Если значение оставлено пустым, то будет сохранена пустая строка.

Установка библиотеки Pillow и раздача медиафайлов

Для работы с изображениями необходимо установить библиотеку `Pillow`. Библиотека `Pillow` – это де-факто стандартная библиотека, предназначенная для обработки изображений на Python. В ней поддерживаются многочисленные форматы изображений и предоставляются мощные функции обработки изображений. Библиотека `Pillow` требуется веб-фреймворку Django для работы с изображениями в классе `ImageField`.

Установите `Pillow`, выполнив следующую ниже команду из командной оболочки:

```
pip install Pillow==9.2.0
```

Отредактируйте файл `settings.py` проекта, добавив следующие ниже строки:

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

Эти настройки обеспечат Django возможность управлять закачиванием файлов на сайт и раздачей медиафайлов. `MEDIA_URL` – это базовый URL-адрес, используемый для раздачи медиафайлов, закачанных пользователями на сайт. `MEDIA_ROOT` – это локальный путь, где они находятся. Пути и URL-адреса файлов формируются динамически посредством добавления к ним пути проекта или URL-адреса медиафайлов в качестве префикса с целью переносимости.

Теперь отредактируйте главный файл `urls.py` проекта `bookmarks`, видоизменив исходный код, как показано ниже. Новые строки выделены жирным шрифтом:

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

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

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

Была добавлена вспомогательная функция `static()`, чтобы раздавать медиафайлы с помощью сервера разработки во время разработки (то есть когда настроечный параметр `DEBUG` задан равным `True`).

Вспомогательная функция `static()` подходит только для разработки, но не для использования в производстве. Django очень неэффективен при раздаче статических файлов. Никогда не раздавайте статические файлы с помощью Django в производственной среде. В главе 17 «Выход в прямой эфир» вы научитесь раздавать статические файлы в производственной среде.

Создание миграций для модели профиля

Откройте оболочку и выполните следующую ниже команду, чтобы создать миграции базы данных для новой модели:

```
python manage.py makemigrations
```

Вы получите такой результат:

```
Migrations for 'account':
  account/migrations/0001_initial.py
    - Create model Profile
```

Затем следующей ниже командой командной оболочки синхронизируйте базу данных:

```
python manage.py migrate
```

Вы увидите результат, включающий такую строку:

```
Applying account.0001_initial... OK
```

Отредактируйте файл `admin.py` приложения `account`, чтобы зарегистрировать модель `Profile` на сайте администрирования, добавив исходный код, выделенный жирным шрифтом:

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

@admin.register(Profile)
class ProfileAdmin(admin.ModelAdmin):
    list_display = ['user', 'date_of_birth', 'photo']
    raw_id_fields = ['user']
```

Следующей ниже командой запустите сервер разработки из командной оболочки:

```
python manage.py runserver
```

Пройдите по URL-адресу `http://127.0.0.1:8000/admin/` в своем браузере. Сейчас вы должны увидеть модель `Profile` на сайте администрирования проекта, как показано ниже:

*Рис. 4.19. Блок **ACCOUNT** (Учетная запись) на индексной странице сайта администрирования*

Кликните по ссылке **Add** (Добавить) в строке **Profiles** (Профили). Вы увидите следующую ниже форму для добавления нового профиля:

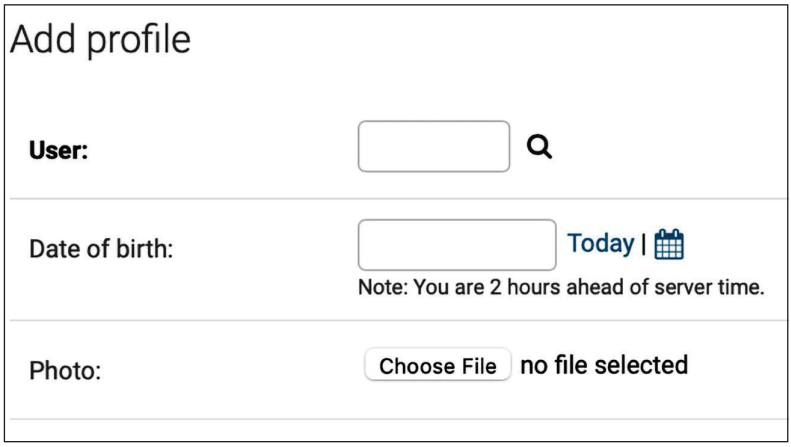

Рис. 4.20. Форма для добавления профиля

Создайте объект `Profile` вручную для каждого существующего пользователя в базе данных.

Далее мы предоставим пользователям возможность редактировать свои профили на сайте.

Отредактируйте файл `forms.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
# ...
from .models import Profile

# ...

class UserEditForm(forms.ModelForm):
    class Meta:
            model = User
            fields = ['first_name', 'last_name', 'email']

class ProfileEditForm(forms.ModelForm):
    class Meta:
            model = Profile
            fields = ['date_of_birth', 'photo']
```

Эти формы таковы:

- `UserEditForm` позволит пользователям редактировать свое имя, фамилию и адрес электронной почты, которые являются атрибутами встроенной в Django модели `User`;
- `ProfileEditForm` позволит пользователям редактировать данные профиля, сохраненные в конкретно-прикладной модели `Profile`. Пользователи смогут редактировать дату своего рождения и закачивать изображение на сайт в качестве фотоснимка профиля.

Отредактируйте файл `views.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
# ...
from .models import Profile

# ...

def register(request):
    if request.method == 'POST':
        user_form = UserRegistrationForm(request.POST)
        if user_form.is_valid():
            # Создать новый объект пользователя,
            # но пока не сохранять его
```

```
            new_user = user_form.save(commit=False)
            # Установить выбранный пароль
            new_user.set_password(
                user_form.cleaned_data['password'])
            # Сохранить объект User
            new_user.save()
            # Создать профиль пользователя
            Profile.objects.create(user=new_user)
            return render(request,
                          'account/register_done.html',
                          {'new_user': new_user})
    else:
        user_form = UserRegistrationForm()
    return render(request,
                  'account/register.html',
                  {'user_form': user_form})
```

При регистрации пользователей в системе будет создаваться объект `Profile`, который будет ассоциирован с созданным объектом `User`.

Теперь мы предоставим пользователям возможность редактировать свои профили.

Отредактируйте файл `views.py` приложения `account`, добавив следующий ниже исходный код, выделенный жирным шрифтом:

```
from django.http import HttpResponse
from django.shortcuts import render
from django.contrib.auth import authenticate, login
from django.contrib.auth.decorators import login_required
from .forms import LoginForm, UserRegistrationForm, \
                   UserEditForm, ProfileEditForm
from .models import Profile

# ...

@login_required
def edit(request):
    if request.method == 'POST':
        user_form = UserEditForm(instance=request.user,
                                 data=request.POST)
        profile_form = ProfileEditForm(
                                    instance=request.user.profile,
                                    data=request.POST,
                                    files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
            profile_form.save()
    else:
```

```
        user_form = UserEditForm(instance=request.user)
        profile_form = ProfileEditForm(
                                    instance=request.user.profile)
    return render(request,
                  'account/edit.html',
                  {'user_form': user_form,
                   'profile_form': profile_form})
```

Мы добавили новое представление `edit`, чтобы пользователи могли редактировать свою личную информацию. Мы добавили в него декоратор `login_required`, поскольку только аутентифицированные пользователи могут редактировать свои профили. В этом представлении используются две модельные формы: `UserEditForm` для хранения данных во встроенной модели `User` и `ProfileEditForm` для хранения дополнительных персональных данных в конкретно-прикладной модели `Profile`. В целях валидации переданных данных вызывается метод `is_valid()` обеих форм. Если обе формы содержат валидные данные, то обе формы сохраняются путем вызова метода `save()`, чтобы обновить соответствующие объекты в базе данных.

Добавьте следующий ниже шаблон URL-адреса в файл `urls.py` приложения `account`:

```
urlpatterns = [
    #...
    path('', include('django.contrib.auth.urls')),
    path('', views.dashboard, name='dashboard'),
    path('register/', views.register, name='register'),
    path('edit/', views.edit, name='edit'),
]
```

Наконец, создайте шаблон данного представления, разместив его в каталоге `templates/account/`, и назовите этот шаблон `edit.html`. Добавьте в него следующий ниже исходный код:

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

{% block title %}Edit your account{% endblock %}

{% block content %}
  <h1>Edit your account</h1>
  <p>You can edit your account using the following form:</p>
  <form method="post" enctype="multipart/form-data">
    {{ user_form.as_p }}
    {{ profile_form.as_p }}
    {% csrf_token %}
    <p><input type="submit" value="Save changes"></p>
  </form>
{% endblock %}
```

В приведенном выше исходном коде был добавлен в HTML-элемент `<form>` атрибут `enctype="multipart/form-data"`, чтобы обеспечить закачивание файлов на сайт. Мы используем HTML-форму для передачи форм `user_form` и `profile_form` на обработку.

Откройте URL-адрес `http://127.0.0.1:8000/account/register/` и зарегистрируйте нового пользователя. Затем войдите под новым пользователем и откройте URL-адрес `http://127.0.0.1:8000/account/edit/`. Вы должны увидеть следующую ниже страницу:

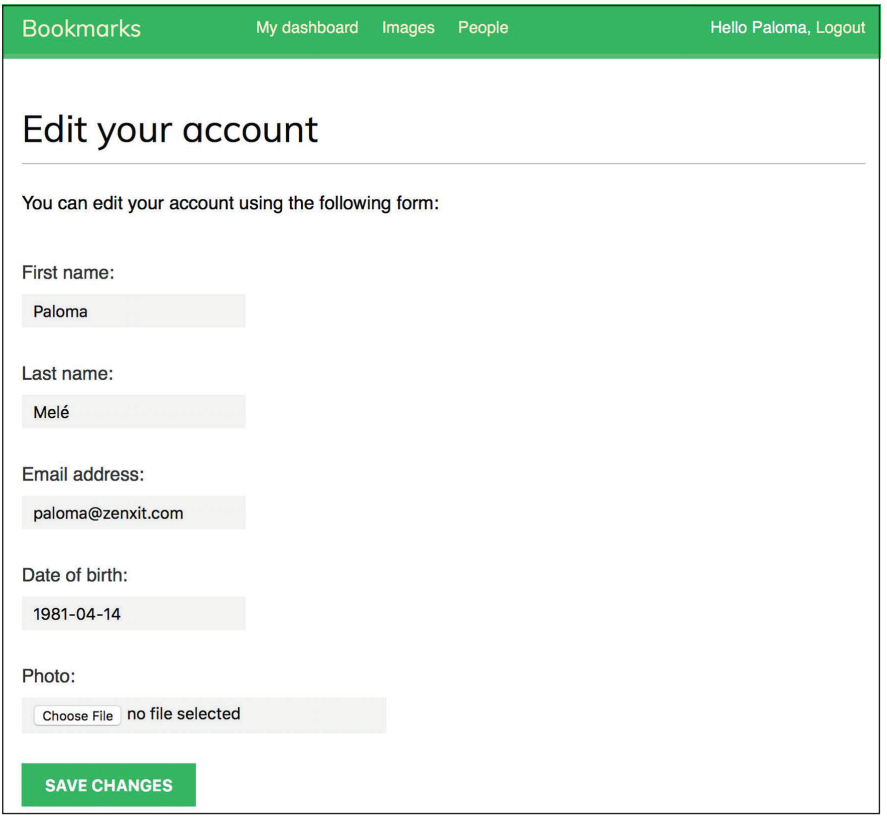

Рис. 4.21. Форма для редактирования профиля

Теперь можно добавлять информацию о профиле и сохранять изменения.

Далее мы отредактируем шаблон информационной панели, вставив в него ссылки на страницы редактирования профиля и смены пароля.

Откройте шаблон `templates/account/dashboard.html` и добавьте следующие ниже строки, выделенные жирным шрифтом:

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

{% block title %}Dashboard{% endblock %}
```

```
{% block content %}
  <h1>Dashboard</h1>
  <p>
    Welcome to your dashboard. You can <a href="{% url 'edit' %}">edit your profile</a>
or <a href="{% url 'password_change' %}">change your password</a>.
  </p>
{% endblock %}
```

Теперь пользователи могут обращаться к форме редактирования своего профиля из информационной панели. Пройдите по URL-адресу `http://127.0.0.1:8000/account/` в своем браузере и проверьте новую ссылку на редактирование профиля пользователя. Теперь информационная панель должна выглядеть следующим образом:

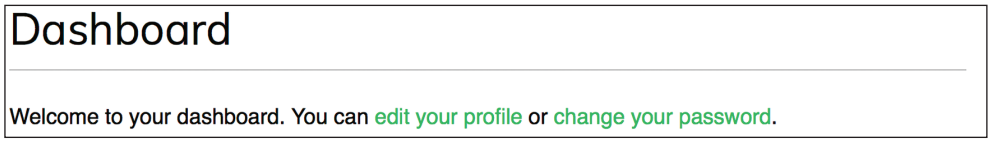

Рис. 4.22. Содержимое страницы информационной панели, включающей ссылки на редактирование профиля и смены пароля

Использование конкретно-прикладной модели пользователя

Django также предлагает способ замены модели `User` на конкретно-прикладную модель. Класс `User` должен наследовать от класса `AbstractUser` веб-фреймворка Django, который предоставляет полную реализацию стандартного пользователя в виде абстрактной модели. Подробнее об этом методе можно прочитать по адресу https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#substituting-a-custom-user-model.

Использование конкретно-прикладной модели пользователя будет придавать больше гибкости, но оно также может приводить к более сложной интеграции с подключаемыми приложениями, которые взаимодействуют с моделью пользователя `auth` веб-фреймворка Django напрямую.

Использование фреймворка сообщений

При взаимодействии пользователей с платформой нередко возникает потребность информировать их о результате определенных действий. Django имеет встроенный фреймворк сообщений, который позволяет показывать пользователям одноразовые уведомления.

Фреймворк сообщений находится в `django.contrib.messages` и вставляется в стандартный список `INSTALLED_APPS` в файле `settings.py` при создании новых проектов с помощью команды `python manage.py startproject`. В параметре

MIDDLEWARE файла настроечных параметров settings.py также содержатся промежуточные программные компоненты django.contrib.messages.middleware.MessageMiddleware.

Фреймворк сообщений предоставляет пользователям простой способ добавления сообщений. По умолчанию сообщения хранятся в cookie-файле (отступая к сеансовому хранению), и они отображаются и очищаются при следующем запросе от пользователя. Фреймворк сообщений можно использовать в своих представлениях, импортируя модуль messages и добавляя новые сообщения с помощью простых методов сокращенного доступа, как показано ниже:

```
from django.contrib import messages
messages.error(request, 'Something went wrong')
```

Новые сообщения можно создавать, используя метод add_message() или любой метод сокращенного доступа, приведенный ниже:

- success(): сообщения об успехе, отображаемые при успешном выполнении действия;
- info(): информационные сообщения;
- warning(): сбой еще не произошел, но он, возможно, неизбежен;
- error(): действие не было успешным либо произошел сбой;
- debug(): отладочные сообщения, которые будут удалены либо проигнорированы в производственной среде.

Давайте добавим сообщения в проект. Фреймворк сообщений применяется к проекту глобально. Для отображения всех имеющихся для клиента сообщений мы будем использовать базовый шаблон. Такой подход позволит уведомлять клиента о результатах любого действия на любой странице.

Откройте шаблон templates/base.html приложения account и добавьте в него следующий ниже исходный код, выделенный жирным шрифтом:

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
  <title>{% block title %}{% endblock %}</title>
  <link href="{% static "css/base.css" %}" rel="stylesheet">
</head>
<body>
  <div id="header">
    ...
  </div>
  {% if messages %}
    <ul class="messages">
      {% for message in messages %}
        <li class="{{ message.tags }}">
          {{ message|safe }}
```

```
            <a href="#" class="close">x</a>
          </li>
        {% endfor %}
      </ul>
    {% endif %}
    <div id="content">
      {% block content %}
      {% endblock %}
    </div>
  </body>
  </html>
```

В фреймворке сообщений содержится процессор контекста под названием `django.contrib.messages.context_processors.messages`, который добавляет переменную `messages` в контекст запроса. Он находится в списке `context_processors` в настроечном параметре `TEMPLATES` проекта. Переменная `messages` используется в шаблонах для отображения пользователю всех существующих сообщений.

Процессор контекста – это функция Python, которая принимает объект `request` в качестве аргумента и возвращает словарь, который добавляется в контекст запроса. Вы научитесь создавать свои собственные процессоры контекста в главе 8 «Разработка интернет-магазина».

Давайте видоизменим представление редактирования, чтобы использовать фреймворк сообщений.

Отредактируйте файл `views.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом:

```
# ...
from django.contrib import messages

# ...

@login_required
def edit(request):
    if request.method == 'POST':
        user_form = UserEditForm(instance=request.user,
                                 data=request.POST)
        profile_form = ProfileEditForm(
                                    instance=request.user.profile,
                                    data=request.POST,
                                    files=request.FILES)
        if user_form.is_valid() and profile_form.is_valid():
            user_form.save()
```

```
            profile_form.save()
            messages.success(request, 'Profile updated '\
                                      'successfully')
        else:
            messages.error(request, 'Error updating your profile')
    else:
        user_form = UserEditForm(instance=request.user)
        profile_form = ProfileEditForm(
                                    instance=request.user.profile)
    return render(request,
                  'account/edit.html',
                  {'user_form': user_form,
                   'profile_form': profile_form})
```

Сообщение об успехе генерируется, когда пользователи успешно обновляют свой профиль. Если в какой-либо из форм содержатся невалидные данные, то вместо него генерируется сообщение об ошибке.

Пройдите по URL-адресу `http://127.0.0.1:8000/account/edit/` в своем браузере и отредактируйте профиль пользователя. При успешном обновлении профиля вы должны увидеть следующее ниже сообщение:

Рис. 4.23. Сообщение об успешно отредактированном профиле

Введите недействительную дату в поле **Date of birth** (Дата рождения) и снова передайте форму на обработку. Вы должны увидеть следующее сообщение:

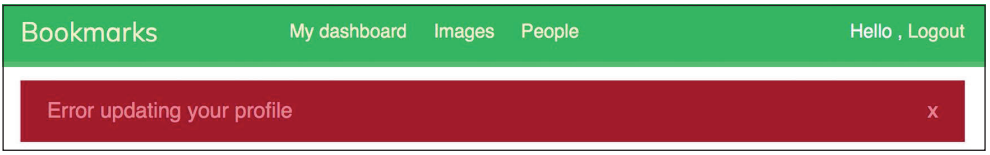

Рис. 4.24. Сообщение об ошибке обновления профиля

Сообщения, которые информируют пользователей о результатах их действий, генерируются по-настоящему просто, причем сообщения можно легко добавлять и в другие представления.

Подробнее о фреймворке сообщений можно узнать на странице https://docs.djangoproject.com/en/4.1/ref/contrib/messages/.

Теперь, когда мы разработали всю функциональность, связанную с аутентификацией пользователей и редактированием профиля, мы углубимся в адаптацию процедуры аутентификации под конкретно-прикладную задачу. Мы научимся разрабатывать конкретно-прикладной бэкенд аутентификации, чтобы пользователи имели возможность входить в систему, используя свой адрес электронной почты.

Разработка конкретно-прикладного бэкенда аутентификации

Django позволяет аутентифицировать пользователей по различным источникам. Настроечный параметр `AUTHENTICATION_BACKENDS` содержит список доступных в проекте бэкендов аутентификации. По умолчанию этот настроечный параметр имеет следующее значение:

```
['django.contrib.auth.backends.ModelBackend'].
```

Применяемый по умолчанию бэкенд `ModelBackend` аутентифицирует пользователей по базе данных, используя модель `User` из `django.contrib.auth`. Такой подход приемлем для большинства веб-проектов. Однако еще есть возможность создавать конкретно-прикладные бэкенды, чтобы аутентифицировать пользователей по другим источникам, например по каталогу на основе протокола облегченного доступа к каталогам (**LDAP**)[1] или любой другой системе.

Более подробную информацию об адаптации процедуры аутентификации под конкретно-прикладную задачу можно почитать по адресу https://docs.djangoproject.com/en/4.1/topics/auth/customizing/#other-authentication-sources.

При использовании функции `authenticate()` из `django.contrib.auth` Django пытается по очереди аутентифицировать пользователя на каждом из бэкендов, определенных в `AUTHENTICATION_BACKENDS`, до тех пор, пока один из них не аутентифицирует пользователя успешно. Пользователь не будет аутентифицирован, только если всем бэкендам не удалось выполнить аутентификацию.

Django предоставляет простой способ определения своих собственных бэкендов аутентификации. Бэкенд аутентификации – это класс, который предоставляет следующие два метода:

- `authenticate()`: принимает объект `request` и учетные данные пользователя в качестве параметров. Он возвращает объект `user`, соответствующий этим учетным данным, если учетные данные валидны, либо `None` в противном случае. Параметр `request` – это объект `HttpRequest`, либо `None`, если он не передан функции `authenticate()`;

[1] Англ. Lightweight Directory Access Protocol. – *Прим. перев.*

- `get_user()`: принимает ИД пользователя в качестве параметра и возвращает объект `user`.

Конкретно-прикладной бэкенд аутентификации создается так же просто, как и класс Python, реализующий оба метода. Давайте создадим бэкенд аутентификации, который позволит пользователям аутентифицироваться в системе, используя адрес электронной почты вместо пользовательского имени (username).

Внутри каталога приложения `account` создайте новый файл и назовите его `authentication.py`. Добавьте в него следующий ниже исходный код:

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

class EmailAuthBackend:
"""
Аутентифицировать посредством адреса электронной почты.
"""
def authenticate(self, request, username=None, password=None):
    try:
        user = User.objects.get(email=username)
        if user.check_password(password):
            return user
        return None
    except (User.DoesNotExist, User.MultipleObjectsReturned):
        return None

    def get_user(self, user_id):
        try:
            return User.objects.get(pk=user_id)
        except User.DoesNotExist:
            return None
```

Приведенный выше исходный код является простым бэкендом аутентификации. Метод `authenticate()` получает объект `request` и опциональные параметры `username` и `password`. Мы могли бы использовать другие параметры, но мы используем `username` и `password`, чтобы этот бэкенд сразу же заработал с представлениями из фреймворка аутентификации. Показанный выше исходный код работает следующим образом:

- `authenticate()`: извлекается пользователь с данным адресом электронной почты, а пароль проверяется посредством встроенного метода `check_password()` модели пользователя. Указанный метод хеширует пароль, чтобы сравнить данный пароль с паролем, хранящимся в базе данных. Отлавливаются два разных исключения, относящихся к набору запросов QuerySet: `DoesNotExist` и `MultipleObjectsReturned`. Исключение

`DoesNotExist` возникает, если пользователь с данным адресом электронной почты не найден. Исключение `MultipleObjectsReturned` возникает, если найдено несколько пользователей с одним и тем же адресом электронной почты. Позже мы видоизменим представления регистрации и редактирования, чтобы предотвратить использование пользователями существующего адреса электронной почты;

- `get_user()`: пользователь извлекается по его ИД, указанному в параметре `user_id`. Django использует аутентифицировавший пользователя бэкенд, чтобы извлечь объект `User` на время сеанса пользователя. **pk** (сокращение от **primary key**) является уникальным идентификатором каждой записи в базе данных. Каждая модель Django имеет поле, которое служит ее первичным ключом. По умолчанию первичным ключом является автоматически генерируемое поле `id`. Во встроенном в Django ORM-преобразователе первичный ключ тоже может называться pk. Более подробная информация об автоматических полях первичного ключа находится по адресу https://docs.djangoproject.com/en/4.1/topics/db/models/#automatic-primary-key-fields.

Отредактируйте файл `settings.py` проекта, добавив следующий ниже исходный код:

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'account.authentication.EmailAuthBackend',
]
```

В данном настроечном параметре мы оставляем стандартный `ModelBackend`, который используется для аутентификации с помощью пользовательского имени и пароля, и вставляем наш собственный бэкенд аутентификации с применением электронной почты `EmailAuthBackend`.

Пройдите по URL-адресу `http://127.0.0.1:8000/account/login/` в своем браузере. Напомним, что Django будет пытаться аутентифицировать пользователя на каждом бэкенде, поэтому теперь вы должны иметь возможность беспрепятственно входить в систему, используя свое пользовательское имя либо учетную запись электронной почты.

Учетные данные пользователя будут проверены с помощью `ModelBackend`, а если пользователь не возвращен, то учетные данные будут проверены с помощью `EmailAuthBackend`.

Порядок следования бэкендов, перечисленных в настроечном параметре `AUTHENTICATION_BACKENDS`, имеет значение. Если одни и те же учетные данные валидны для нескольких бэкендов, то Django остановится на первом бэкенде, который успешно аутентифицирует пользователя.

Предотвращение использования существующего адреса электронной почты

Модель `User` в фреймворке аутентификации не препятствует созданию пользователей с одинаковым адресом электронной почты. Если две или более учетных записей пользователей имеют один и тот же адрес электронной почты, то мы не сможем определить, кто из пользователей проходит аутентификацию. Теперь, когда пользователи могут входить в систему, используя свой адрес электронной почты, мы должны предотвратить регистрацию пользователей с существующим адресом электронной почты.

Сейчас мы изменим форму для регистрации пользователей, чтобы предотвратить регистрацию нескольких пользователей с одним и тем же адресом электронной почты.

Отредактируйте файл `forms.py` приложения `account`, добавив следующие ниже строки, выделенные жирным шрифтом, в класс `UserRegistrationForm`:

```
class UserRegistrationForm(forms.ModelForm):
    password = forms.CharField(label='Password',
                               widget=forms.PasswordInput)
    password2 = forms.CharField(label='Repeat password',
                                widget=forms.PasswordInput)

    class Meta:
        model = User
        fields = ['username', 'first_name', 'email']

    def clean_password2(self):
        cd = self.cleaned_data
        if cd['password'] != cd['password2']:
            raise forms.ValidationError('Passwords don\'t match.')
        return cd['password2']

    def clean_email(self):
        data = self.cleaned_data['email']
        if User.objects.filter(email=data).exists():
            raise forms.ValidationError('Email already in use.')
        return data
```

Мы добавили валидацию поля электронной почты, которая не позволяет пользователям регистрироваться с уже существующим адресом электронной почты. Мы формируем набор запросов QuerySet, чтобы свериться, нет ли существующих пользователей с одинаковым адресом электронной почты. Мы проверяем наличие результатов посредством метода `exists()`. Метод `exists()` возвращает `True`, если набор запросов QuerySet содержит какие-либо результаты, и `False` в противном случае.