­[Урок 1](#Lesson_1)

[Урок 2](#Lesson_2)

[Аутентифікація](#Аутентифікація)

[Профіль користувача](#Профіль_користувача)

[Скидання паролю](#Reset_password)

[Приховані змінні середовища](#Hidden_variables)

Встановлення Django може відбуватися з публічного репозиторію PyPi командою:

pip install django

Щоб перевірити, яка версію встановлена, можна виконати

python -m django --version

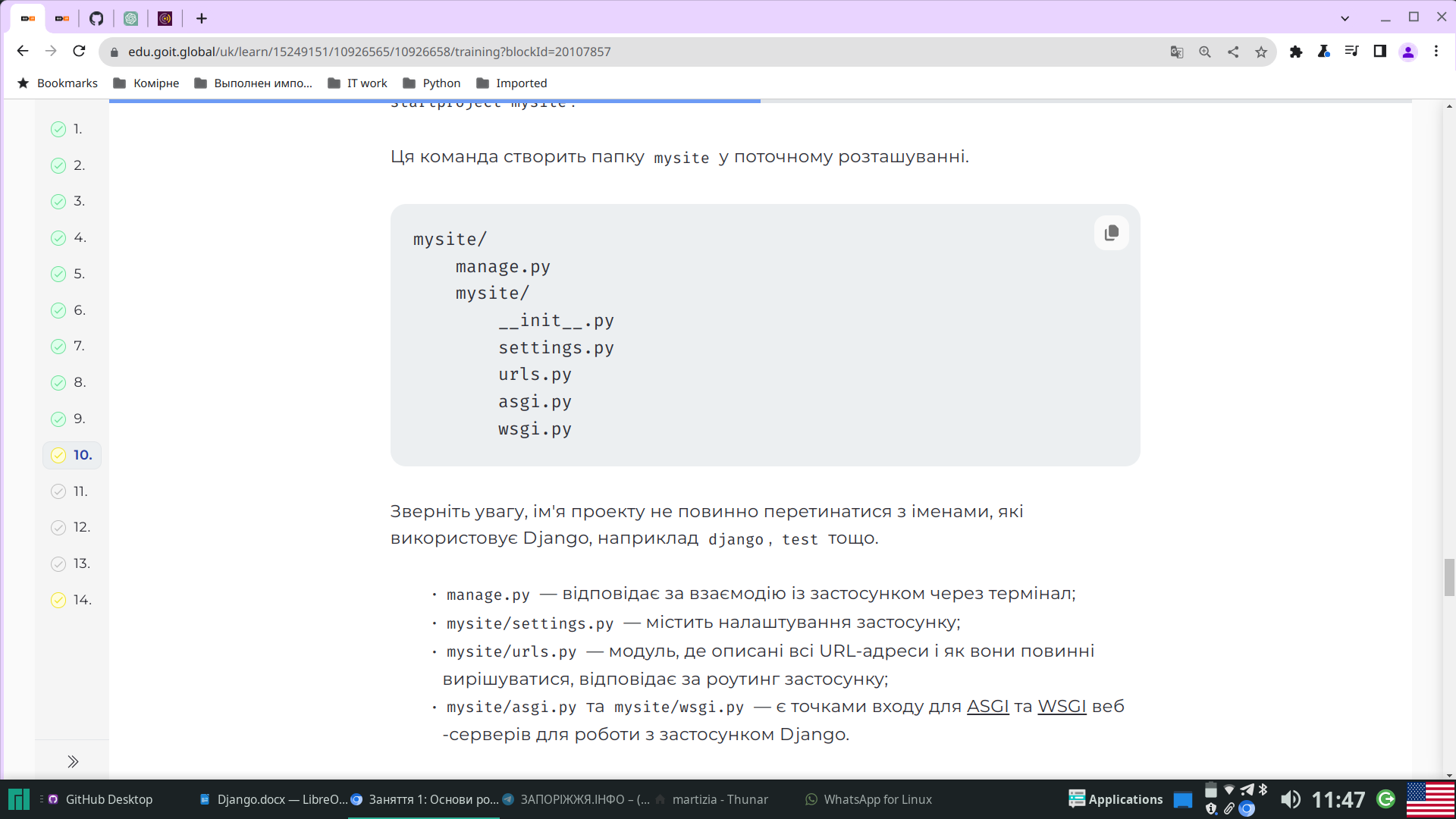
**CAUTION**

*Зверніть увагу, що Django не гарантує зворотну сумісність між версіями. Застосунок, написаний для Django 3.0, може не працювати з версією 3.2.*

Для створення проекту достатньо виконати в папці з проектом

django-admin startproject mysite

Ця команда створить папку mysite у поточному розташуванні.

* manage.py – відповідає за взаємодію із застосунком через термінал;
* mysite/settings.py – містить налаштування застосунку;
* mysite/urls.py – модуль, де описані всі URL-адреси і як вони повинні вирішуватися, відповідає за роутинг застосунку;
* mysite/asgi.py та mysite/wsgi.py – є точками входу для [ASGI](https://en.wikipedia.org/wiki/Asynchronous_Server_Gateway_Interface) та [WSGI](https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface) веб-серверів для роботи з застосунком Django.

Щоб запустити локально сервер для розробки, достатньо виконати команду в консолі

python manage.py runserver

знаходячись всередині папки проекту mysite.

**INFO**

*Цей сервер не підходить для використання в реальних застосунках, але дуже корисний на етапі розробки.*

**Створення web-застосунку**

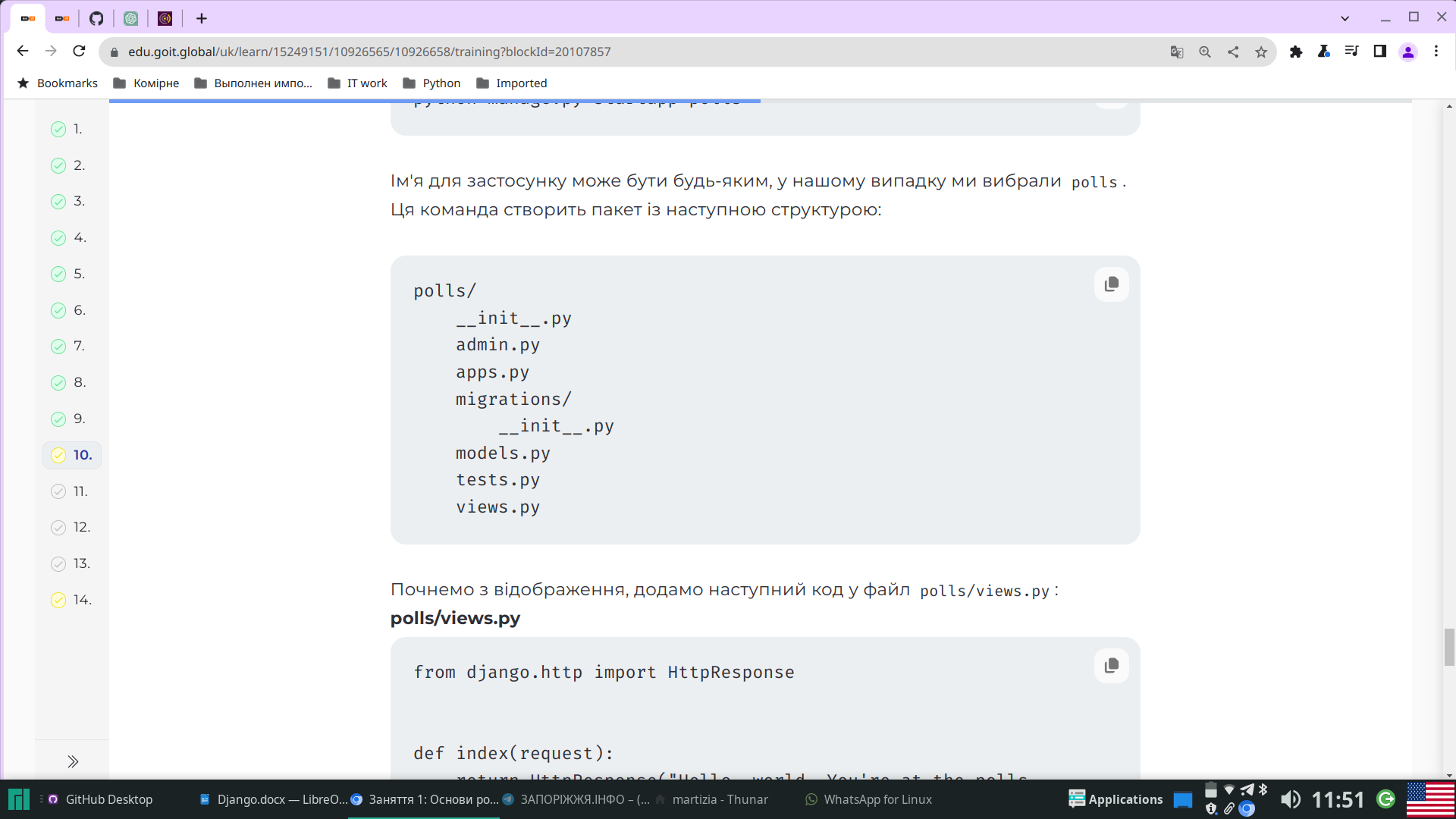
Застосунок – це сутність, яка робить щось конкретне (отримує дані від користувача, відображає інформацію з бази даних тощо). Кожен проект може складатися з кількох застосунків. Застосунки також можуть бути одночасно частиною кількох проектів.

Тепер, коли у нас є проект, ми можемо створити перший застосунок:

python manage.py startapp polls

Ім'я для застосунку може бути будь-яким, у нашому випадку ми вибрали pools.

Ця команда створить пакет із наступною структурою:



Почнемо з відображення, додамо наступний код у файл polls/views.py:

**polls/views.py**

from django.http import HttpResponse

def index(request):

**return HttpResponse("Hello, world. You're at the polls index.")**

Ми імпортували спеціальний клас HttpResponse, який формує відповідь для клієнта. У нашому випадку це текстовий рядок `Hello, world. You're at the polls index.'

Далі, щоб представлення стало видимим, його потрібно зареєструвати. Для цього створимо модуль polls/urls.py

**ВАЖЛИВО**!

*Імена модулів мають значення. Так, Django автоматично зрозуміє, що містить urls.py і як з ним працювати.*

**polls/urls.py**

from django.urls import path

from . import views

urlpatterns = [

path('', views.index, name='index'),

**]**

Ми визначаємо список urlpatterns, в якому будуть маршрути для нашого застосунку. Поки що тут тільки один маршрут під час звернення до застосунку polls і він оброблятиметься функцією index з файлу views.py. Цьому маршруту ми даємо ім'я index. Тепер кореневий каталог програми polls буде дозволений за допомогою views.index.

Щоб зареєструвати сам застосунок polls, у проекті потрібно модифікувати mysite/urls.py:

**mysite/urls.py**

from django.contrib import admin

from django.urls import include, path

urlpatterns = [

path('polls/', include('polls.urls')),

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

]

Тут вже є шлях до адмін-панелі нашого проекту — admin/, і ми додали шлях до створеного застосунку: polls/. Функція include потрібна, щоб повідомити Django, що всі маршрути, що починаються з polls/, повинні оброблятися застосунком polls. Сам маршрут втрачає префікс polls (відбудеться заміна polls/ на /) і відправляється для обробки вже в сам застосунок.

Щоб переглянути результат роботи нашого застосунку, можна виконати python manage.py runserver та перейти на http://localhost:8000/polls/.

**Робота з базами даних в Django**

Django містить дуже потужну ORM з інструментом міграцій. Щоб синхронізувати стан моделей у Python та таблиць у базі даних, достатньо виконати команду

python manage.py migrate

Команда migrate "дивиться" в INSTALLED\_APPS в mysite/settings.py і створює всі необхідні таблиці в базі даних для всіх застосунків.

Давайте створимо моделі для нашого застосунку polls. Для цього перепишемо модуль polls/models.py:

**polls/models.py**

from django.db import models

class Question(models.Model):

question\_text = models.CharField(max\_length=200)

pub\_date = models.DateTimeField('date published')

class Choice(models.Model):

question = models.ForeignKey(Question, on\_delete=models.CASCADE)

choice\_text = models.CharField(max\_length=200)

**votes = models.IntegerField(default=0)**

Кожен клас в polls/models.py отримає своє представлення у базі даних. Атрибути цього класу стануть полями, а об'єкт класу — записом у таблиці.

Ми створили дві моделі для питань Question та для відповідей користувачів Choice.

Тепер потрібно зареєструвати застосунок у проекті, щоб інструмент міграцій знав про ці моделі. Для цього потрібно додати наш застосунок polls до списку застосунків - константа INSTALLED\_APPS у файлі mysite/settings.py. Наша константа INSTALLED\_APPS виглядатиме:

INSTALLED\_APPS = [

"django.contrib.admin",

"django.contrib.auth",

"django.contrib.contenttypes",

"django.contrib.sessions",

"django.contrib.messages",

"django.contrib.staticfiles",

"polls",

]

Для створення міграції для нашого застосунку:

python manage.py makemigrations polls

В результаті буде створено файл міграції polls/migrations/0001\_initial.py з описом таблиць, які потрібно створити у базі даних.

Migrations for 'polls':

polls\migrations\0001\_initial.py

- Create model Question

- Create model Choice

Міграції ще не застосовувалися, ви можете подивитися, що повинно бути зроблено в міграціях, та змінити цей файл, якщо потрібно.

Щоб застосувати всі міграції:

python manage.py migrate

Щоб змінити налаштування роботи з базою даних, потрібно змінити значення константи DATABASES у файлі mysite/settings.py. Django налаштований на роботу з SQLite за замовчуванням.

DATABASES = {

"default": {

"ENGINE": "django.db.backends.sqlite3",

"NAME": BASE\_DIR / "db.sqlite3",

}

}

**INFO**

*Щоб змінити, як саме Django відображатиме об'єкт моделі, достатньо змінити метод \_\_str\_\_ для класу моделі.*

Щоб показати, як використовувати моделі у представленні, змінимо файл polls/views.py:

**polls/views.py**

from django.http import HttpResponse

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

output = ', '.join([q.question\_text for q in latest\_question\_list])

**return HttpResponse(output)**

Тепер застосунок polls відображає клієнту останні 5 питань при запиті на кореневий ресурс застосунку /.

**ПРИМІТКА**

*Але самі дані в базі даних поки що відсутні, і щоб їх додати туди, потрібна адмін-панель.*

**Admin панель в Django**

Щоб користуватися адміністративною панеллю Django, потрібно спочатку створити користувача, який зможе зайти в адмін-панель:

python manage.py createsuperuser

Після того, як користувач успішно створений, ви можете запустити сервер і зайти на http://127.0.0.1:8000/admin/, використовуючи ім'я та пароль для нового користувача.

Поки що в панелі адміністратора немає нічого для роботи із застосунком polls. Давайте зареєструємо таблиці Question та Choice в адмін-панелі. Для цього у файлі polls/admin.py додамо наступні рядки:

from django.contrib import admin

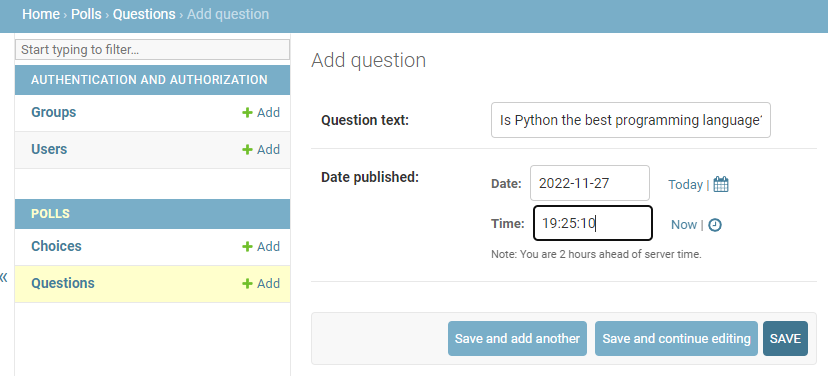
from .models import Question, Choice

admin.site.register(Question)

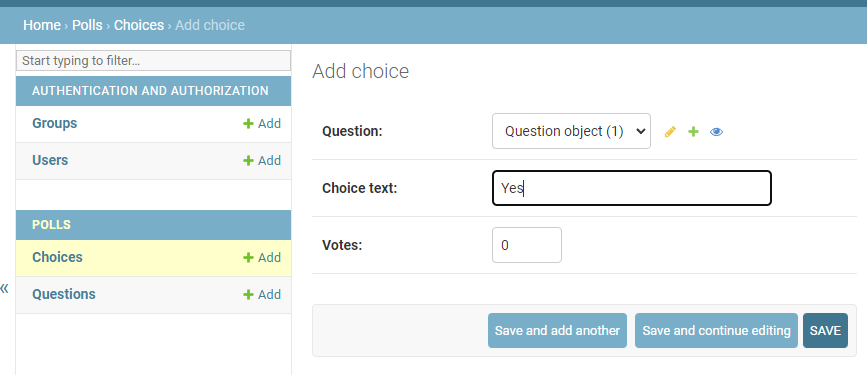
admin.site.register(Choice)

Тепер ви можете додавати питання для застосунку polls, використовуючи адмін-панель.

Додамо перше запитання "Is Python the best programming language?":



А потім додамо дві відповіді **Yes** та **No**



**Аргументи запиту**

Давайте додамо ще представлень у файл polls/views.py:

def detail(request, question\_id):

return HttpResponse("You're looking at question %s." % question\_id)

def results(request, question\_id):

response = "You're looking at the results of question %s."

return HttpResponse(response % question\_id)

def vote(request, question\_id):

return HttpResponse("You're voting on question %s." % question\_id)

І зареєструємо ці представлення у файлі polls/urls.py:

from django.urls import path

from . import views

app\_name = 'polls'

urlpatterns = [

# ex: /polls/

path('', views.index, name='index'),

# ex: /polls/5/

path('<int:question\_id>/', views.detail, name='detail'),

# ex: /polls/5/results/

path('<int:question\_id>/results/', views.results, name='results'),

# ex: /polls/5/vote/

path('<int:question\_id>/vote/', views.vote, name='vote'),

]

Зверніть увагу на аргументи запиту, визначені у форматі <int:question\_id>/.

Тепер шлях /polls/34/ буде оброблений представленням views.detail, куди як аргумент question\_id буде передано число 34. Також ми додали ім'я застосунку app\_name = 'polls'. Це необхідно, щоб працювали конструкції виду <a href="{% url 'polls:detail' question.id %}">{{ question.question\_text }}</a>

**Django шаблони**

Django містить свій інструмент для генерації документів із шаблонів. Синтаксис створення шаблонів дуже схожий на такий у Jinja, з яким ви вже знайомі, але API відрізняється.

Щоб Django автоматично виявив шаблони застосунку, потрібно в пакеті застосунку створити папку polls/templates/polls. Усі HTML документи у цій папці будуть сприйняті як шаблони. Давайте створимо файл polls/templates/polls/index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Polls</title>

<!-- CSS only -->

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"

rel="stylesheet"

integrity="sha384-rbsA2VBKQhggwzxH7pPCaAqO46MgnOM80zW1RWuH61DGLwZJEdK2Kadq2F9CUG65"

crossorigin="anonymous">

</head>

<body>

{% if latest\_question\_list %}

<ul>

{% for question in latest\_question\_list %}

<li><a href="/polls/{{ question.id }}/">{{ question.question\_text }}</a></li>

{% endfor %}

</ul>

{% else %}

<p>No polls are available.</p>

{% endif %}

</body>

[</html>](https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/)

**INFO**

*Для стилізації шаблонів ми використовуємо популярний фреймворк* [*Bootstrap*](https://getbootstrap.com/)

І оновимо файл polls/views.py таким чином, щоб використати цей шаблон:

from django.http import HttpResponse

from django.template import loader

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

template = loader.get\_template('polls/index.html')

context = {

'latest\_question\_list': latest\_question\_list,

}

return HttpResponse(template.render(context, request))

Або з використанням синтаксичного цукру:

from django.shortcuts import render

from .models import Question

def index(request):

latest\_question\_list = Question.objects.order\_by('-pub\_date')[:5]

context = {'latest\_question\_list': latest\_question\_list}

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

Для спрощення створення посилань у шаблонах, щоб не писати їх хардкодом, можна переписати шаблон:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Polls</title>

<!-- CSS only -->

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css"

rel="stylesheet"

integrity="sha384-rbsA2VBKQhggwzxH7pPCaAqO46MgnOM80zW1RWuH61DG

LwZJEdK2Kadq2F9CUG65" crossorigin="anonymous">

</head>

<body>

<div class="container">

{% if latest\_question\_list %}

<h1>List of questions</h1>

<ul>

{% for question in latest\_question\_list %}

<li>

<a href="{% url 'polls:detail' question.id %}">{{ question.question\_text }}</a>

</li>

{% endfor %}

</ul>

{% else %}

<p>No polls are available.</p>

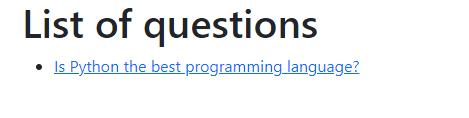
{% endif %}

</div>

</body>

</html>

Вигляд головної сторінки повинен відображати перші 5 запитів, але в нашій базі зараз лише один:

**Обробка форм**

Давайте створимо просту форму для вибору відповіді на запитання. Для цього створимо шаблон форми в polls/templates/polls/detail.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Detail</title>

<!-- CSS only -->

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet"

integrity="sha384-rbsA2VBKQhggwzxH7pPCaAqO46MgnOM80zW1RWuH61DGLwZJEdK2Kadq2F9CUG65" crossorigin="anonymous">

</head>

<body>

<div class="container">

<form action="{% url 'polls:vote' question.id %}" method="post" class="mb-3">

{% csrf\_token %}

<fieldset>

<legend>

<h1>{{ question.question\_text }}</h1>

</legend>

{% if error\_message %}

<p><strong>{{ error\_message }}</strong></p>

{% endif %}

{% for choice in question.choice\_set.all %}

<div class="form-check">

<input

class="form-check-input"

type="radio"

name="choice"

id="choice{{ forloop.counter }}"

value="{{ choice.id }}"

/>

<label for="choice{{ forloop.counter }}" class="form-check-label">{{ choice.choice\_text }}</label>

</div>

<br/>

{% endfor %}

</fieldset>

<button type="submit" class="btn btn-primary"> Vote</button>

</form>

<a href="{% url 'polls:index' %}">Go back to main page</a>

</div>

</body>

</html>

Цей шаблон при виборі відповіді та натисканні кнопки Submit надішле **POST** запит на polls/question.id/vote. Django вміє працювати з [Cross Site Request Forgeries](https://en.wikipedia.org/wiki/Cross-site_request_forgery) і вам не обов'язково налаштовувати цей механізм самостійно, достатньо додати {% csrf\_token %} у шаблон, і токен буде автоматично згенерований та доданий.

Додамо тепер обробник форм у polls/urls.py:

path('<int:question\_id>/vote/', views.vote, name='vote'),

І створимо функцію обробник vote у polls/views.py:

from django.http import HttpResponse, HttpResponseRedirect

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

from django.urls import reverse

from .models import Question, Choice

...

def vote(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

try:

selected\_choice = question.choice\_set.get(pk=request.POST['choice'])

except (KeyError, Choice.DoesNotExist):

# Redisplay the question voting form.

return render(request, 'polls/detail.html', {

'question': question,

'error\_message': "You didn't select a choice.",

})

else:

selected\_choice.votes += 1

selected\_choice.save()

# Always return an HttpResponseRedirect after successfully dealing

# with POST data. This prevents data from being posted twice if a

# user hits the Back button.

return HttpResponseRedirect(reverse('polls:results', args=(question.id,)))

Об'єкт request містить атрибут POST, який поводиться як словник, і ви можете отримати значення полів форми: request.POST['choice']. Оновлення значення поля виконує метод choice\_set, цей метод автоматично створений Django і встановлює зв'язок між Question і Choice об'єктами.

Після оновлення інформації в базі ми можемо надіслати redirect відповідь, щоб браузер автоматично перенаправив користувача на іншу сторінку. Для цього використовується HttpResponseRedirect, який перенаправить на URL, що відповідає за відображення результатів polls:results з аргументом у запиті, що дорівнює question.id. Щоб згенерувати цю URL-адресу, викликаємо функцію reverse.

Додамо обробник результатів у polls/views.py:

def results(request, question\_id):

question = get\_object\_or\_404(Question, pk=question\_id)

return render(request, 'polls/results.html', {'question': question})

І відповідний йому шаблон у polls/templates/polls/results.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Results</title>

<!-- CSS only -->

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet"

integrity="sha384-rbsA2VBKQhggwzxH7pPCaAqO46MgnOM80zW1RWuH61DGLwZJEdK2Kadq2F9CUG65" crossorigin="anonymous">

</head>

<body>

<div class="container">

<h1>{{ question.question\_text }}</h1>

<ul class="list-unstyled">

{% for choice in question.choice\_set.all %}

<li class="mb-3">

{{ choice.choice\_text }} &mdash;

<mark>{{ choice.votes }}</mark>

vote{{ choice.votes|pluralize }}

</li>

{% endfor %}

</ul>

<a href="{% url 'polls:detail' question.id %}" class="btn btn-info">Vote again?</a>

</div>

</body>

</html>

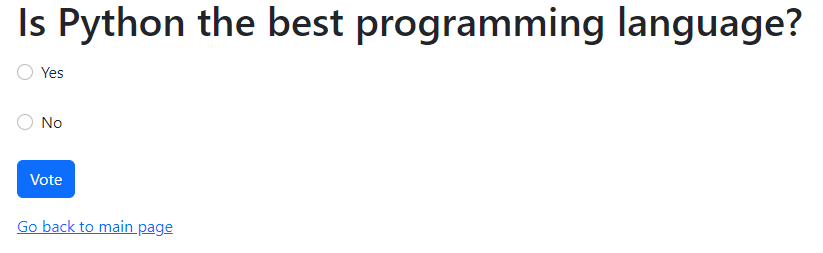
Також потрібно оновити представлення pools/views.py, функцію detail:

def detail(request, question\_id):

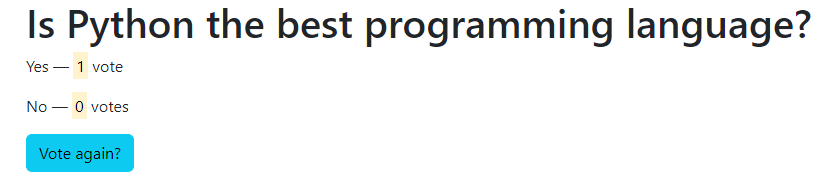
question = get\_object\_or\_404(Question, pk=question\_id)

return render(request, 'polls/detail.html', {'question': question})

Тепер, перейшовши на /polls/1/ у браузері, ви можете проголосувати



та побачити сторінку з результатами після голосування



Ми закінчили вступ до фреймворку Django і розглянули приклад створення базового застосунку. Тепер переходимо до побудови складнішого застосунку.

**Заняття 2**

Створюємо проект Django і називаємо його notes:

django-admin startproject notes

Після виконання команди у нас з'явиться папка notes у кореневій директорії

За допомогою команди cd переходимо до робочої папки проекту notes:

cd notes

Саме звідси ми повинні запускати усі команди Django у терміналі. У самій папці notes знаходиться директорія з такою самою назвою notes та скрипт manage.py.

**Створюємо суперкористувача**

Django поставляється з легковагою базою даних SQLite, яка відмінно підходить для простих проектів, але рідко використовується в продакшені.

**INFO**

*Продакшн — кінцева версія застосунку або сайту, доступна кінцевим користувачам (production). Простіше кажучи, те, що ми можемо знайти в Google, завантажити з Google Play або Apple Store.*

Конфігурація підключення до бази даних знаходиться у файлі settings.py. У проекті ми будемо використовувати сучасну базу даних PostgreSQL.

Запустимо Docker-контейнер, щоб створити сервер PostgreSQL, наступною командою:

docker run --name noteapp-postgres -p 5432:5432 -e POSTGRES\_PASSWORD=567234 -d postgres

Тепер потрібно піти в settings.py та замінити змінну DATABASES. За замовчуванням вона містить підключення до SQLite. Нам необхідно замінити її наступним виразом, щоб підключити PostgreSQL до нашого Django проекту:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql\_psycopg2',

'NAME': 'postgres',

'USER': 'postgres',

'PASSWORD': '567234',

'HOST': '127.0.0.1',

'PORT': '5432',

}

}

Встановимо драйвер psycopg2 для роботи з базою даних PostgreSQL:

poetry add psycopg2

Тепер необхідно створити суперкористувача, але спочатку потрібно застосувати початкову міграцію:

python manage.py migrate

Якщо все пройшло без помилок, ви повинні побачити схоже виведення у терміналі:

Operations to perform:

Apply all migrations: admin, auth, contenttypes, sessions

Running migrations:

Applying contenttypes.0001\_initial... OK

Applying auth.0001\_initial... OK

Applying admin.0001\_initial... OK

Applying admin.0002\_logentry\_remove\_auto\_add... OK

Applying admin.0003\_logentry\_add\_action\_flag\_choices... OK

Applying contenttypes.0002\_remove\_content\_type\_name... OK

Applying auth.0002\_alter\_permission\_name\_max\_length... OK

Applying auth.0003\_alter\_user\_email\_max\_length... OK

Applying auth.0004\_alter\_user\_username\_opts... OK

Applying auth.0005\_alter\_user\_last\_login\_null... OK

Applying auth.0006\_require\_contenttypes\_0002... OK

Applying auth.0007\_alter\_validators\_add\_error\_messages... OK

Applying auth.0008\_alter\_user\_username\_max\_length... OK

Applying auth.0009\_alter\_user\_last\_name\_max\_length... OK

Applying auth.0010\_alter\_group\_name\_max\_length... OK

Applying auth.0011\_update\_proxy\_permissions... OK

Applying auth.0012\_alter\_user\_first\_name\_max\_length... OK

Applying sessions.0001\_initial... OK

Потім виконати команду:

python manage.py createsuperuser

**Перший запуск застосунку**

Запускаємо сервер розробки, щоб розпочати роботу застосунку:

python manage.py runserver

Переходимо за адресою [localhost:8000](http://127.0.0.1:8000/) і бачимо працюючий застосунок.

Щоб потрапити до адмін-панелі для реєстрації як адміністратора переміщуємося на <http://127.0.0.1:8000/admin>

Попереднє налаштування проекту завершено, можна зупинити сервер і перейти до створення застосунку. У терміналі натискаємо на Ctrl+C і припиняємо роботу сервера.

**Створення застосунку noteapp**

У Django для розмежування різних компонентів цілого проекту створюються окремі застосунки. Це рекомендується робити у міру зростання застосунку.

Наприклад, якби ми розробляли систему управління університетом, то в нас би з'явилися застосунки для викладачів, студентів тощо. Наш приклад передбачає створення застосунку з ім'ям noteapp.

python manage.py startapp noteapp

Команда створить папку noteapp в корені проекту.

Тепер необхідно в settings.py додати наш застосунок noteapp в константу INSTALLED\_APPS. У нашому випадку це буде, як показано нижче:

# Application definition

INSTALLED\_APPS = [

"django.contrib.admin",

"django.contrib.auth",

"django.contrib.contenttypes",

"django.contrib.sessions",

"django.contrib.messages",

"django.contrib.staticfiles",

"noteapp",

]

Так **Django** бачить наш застосунок і підключає його до проекту.

**Створення моделей**

Як і інші фреймворки, Django використовує моделі для представлення об'єктів.

Під об'єктами розуміються дані, які підлягають зберіганню у застосунку. У нашому випадку це нотатки, теги тощо.

Відкриваємо файл noteapp/models.py та додаємо наші моделі.

from django.db import models

# Create your models here.

class Tag(models.Model):

name = models.CharField(max\_length=25, null=False, unique=True)

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

return f"{self.name}"

class Note(models.Model):

name = models.CharField(max\_length=50, null=False)

description = models.CharField(max\_length=150, null=False)

done = models.BooleanField(default=False)

created = models.DateTimeField(auto\_now\_add=True)

tags = models.ManyToManyField(Tag)

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

return f"{self.name}"

За допомогою **Django ORM** ми описуємо модель тегів Tag та модель нотаток Note. За допомогою інструкції tags = models.ManyToManyField(Tag) ми створюємо зв'язок багато-до-багатьох між нотатками та тегами. **Django ORM** сам створить для нас проміжну таблицю цих зв'язків.

**Виконання міграцій**

Можливість міграцій вже вбудована у фреймвор **Django**, щоб створити міграцію, потрібно виконати команду:

python manage.py makemigrations

Виведення в термінал:

Migrations for 'noteapp':

noteapp\migrations\0001\_initial.py

- Create model Tag

- Create model Note

Після виконання команди makemigrations отримуємо файл міграцій з інструкціями створення схеми бази даних. Як бачимо з виведення в термінал файл знаходиться на шляху noteapp\migrations\0001\_initial.py. Наступний етап передбачає створення таблиць у БД. Для цієї мети знадобиться команда migrate:

python manage.py migrate

Виведення в термінал:

Operations to perform:

Apply all migrations: admin, auth, contenttypes, noteapp, sessions

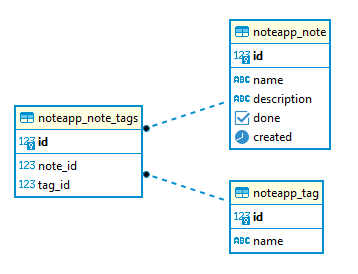
Running migrations:

Applying noteapp.0001\_initial... OK

**НАГАДУВАННЯ**

*Всі команди терміналу ми виконуємо в папці notes. Там, де і знаходиться скрипт manage.py*

Django повинен після цих операцій створити три таблиці у базі даних:



Тепер необхідно виконати реєстрацію моделей у застосунку. Для цього відкриваємо файл noteapp/admin.py та додаємо туди наступний код:

from django.contrib import admin

from .models import Tag, Note

# Register your models here.

admin.site.register(Tag)

admin.site.register(Note)

Робота з налаштування бази даних та визначення моделей у застосунку завершена. Переходимо до реалізації функціоналу нашого застосунку.

**Додавання головної сторінки**

Всередині папки noteapp створимо вкладену папку templates/noteapp, де зберігатимемо шаблони нашого застосунку. Повинно вийти як noteapp/templates/noteapp — така вкладеність - вимога фреймворку Django.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8"/>

<title>Organizer</title>

<link rel="stylesheet" href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"/>

</head>

<body>

<main class="container">

<h1>Main page</h1>

</main>

</body>

</html>

Туди ми помістимо шаблон index.html. Зі зростанням застосунку в цей шаблон додаватиметься додатковий код, але зараз він просто виводить повідомлення "Main page".

Для стилізації html елементів вибрано фреймворк [Pico.css](https://picocss.com/). Його функціоналу цілком вистачить для такого невеликого проекту, як наш. Надалі ми додамо кастомні стилі та побачимо, як підключаються статичні ресурси Django.

Потім підемо і створимо всередині файлу noteapp/views.py функцію main для обробки запитів до застосунку.

from django.shortcuts import render

# Create your views here.

def main(request):

return render(request, 'noteapp/index.html')

Тепер потрібно додати маршрутизацію. Якщо ми заглянемо всередину файлу проекту notes/urls.py, ми побачимо великий коментар, що описує способи створення маршрутів.

У нашому прикладі ми будемо дотримуватися останнього способу організації маршрутів. І тоді файл виглядатиме так:

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path("admin/", admin.site.urls),

path('', include('noteapp.urls')),

]

Залишилося створити файл noteapp/urls.py та додати маршрут.

from django.urls import path

from . import views

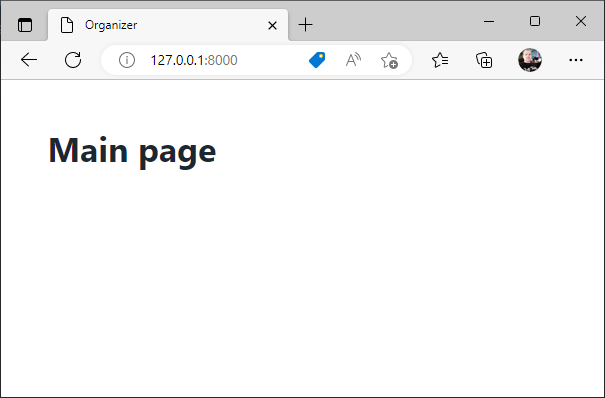
app\_name = 'noteapp'

urlpatterns = [

path('', views.main, name='main'),

]

Запустимо сервер і перейдемо за адресою [`http://localhost:8000/](http://localhost:8000/)`. І побачимо наше повідомлення.



**Додавання тегів**

**Створення шаблону**

**Базовий шаблон**

Оскільки шаблони сторінок нашого застосунку використовуватимуть схожий код, ми спочатку створюємо базовий шаблон base.html, від якого будемо наслідувати інші шаблони.

**notes/noteapp/templates/noteapp/base.html**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8"/>

<title>Note information</title>

<link rel="stylesheet" href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"/>

{% load static %}

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

</head>

<body>

<main class="container">

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

<ul>

<li><a href="/" role="button"> Go back to the main page </a></li>

</ul>

</nav>

{% block content %} {% endblock %}

</main>

</body>

**</html>**

**Статичні ресурси**

Всередині шаблону base.html ми підвантажуємо статичні ресурси:

{% load static %}

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

Всередині файлу налаштувань notes/notes/settings.py знаходиться змінна STATIC\_URL, яка визначає, де зберігатимуться статичні ресурси:

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/4.1/howto/static-files/

STATIC\_URL = "static/"

Давайте створимо файл style.css з наступним вмістом на шляху notes/noteapp/static/noteapp/style.css:

**notes/noteapp/static/noteapp/style.css**

.alert-error, .errorlist {

color: crimson;

}

.alert-success {

color: green;

**}**

Ці стилі ми будемо використовувати для повідомлень у шаблонах: червоний колір crimson ми використовуємо для виведення помилок, а колір green - для виведення успішних операцій.

**Шаблон тегу**

Тепер необхідно додати шаблон noteapp/tag.html у стилі **Django**

**notes/noteapp/templates/noteapp/tag.html**

{% extends "noteapp/base.html" %}

{% block content %}

<form method="POST" action="{% url 'noteapp:tag' %}">

{% csrf\_token %}

<div style="padding: 10px">

<label>Enter the tag name:

{{form.name}}

</label>

<span>{{ form.errors.name }}</span>

</div>

<div class="grid">

<button type="submit">Submit</button>

<button type="reset" class="secondary">Reset</button>

</div>

</form>

{% endblock %}

Інструкція {% extends "noteapp/base.html" %} у шаблоні відповідає за вставлення блоку content всередину базового шаблону base.html.

Ми замінили для форми атрибут action, де кажемо, що потрібно звертатися до маршруту noteapp:tag, який створимо трохи згодом. У самому маршруті є префікс з назвою нашого застосунку noteapp і сама назва маршруту tag.

Також додали інструкцію {% csrf\_token %} для захисту від [міжсайтової підробки запиту](https://en.wikipedia.org/wiki/Cross-site_request_forgery). Це обов'язкова вимога до вмісту форми для фреймворку Django.

Інструкція {{form.name}} відповідає за поле введення тегу. Тут ми використовуємо форму Django form, яку створимо трохи згодом. Замість інструкції {{form.errors.name}}, буде виведено повідомлення про помилку введення, якщо дані нашої форми form виявляться не валідними.

**Створення форми**

Далі нам потрібна сама форма form, яка відображатиметься у шаблоні для користувачів, відповідатиме за коректність введених даних та зберігатиме результат введення в базі даних.

Створимо модуль forms.py всередині застосунку noteapp, не забувши спочатку імпортувати необхідні залежності.

**notes/noteapp/forms.py**

from django.forms import ModelForm, CharField, TextInput

from .models import Tag

class TagForm(ModelForm):

name = CharField(min\_length=3, max\_length=25, required=True, widget=TextInput())

class Meta:

model = Tag

**fields = ['name']**

Форми Django - це клас, поля якого зіставляються з елементами html-форми. Клас forms має різні поля для обробки різних типів даних.

Наприклад, CharField, DateField тощо. Ознайомтеся з документацією [**тут**](https://docs.djangoproject.com/en/3.2/ref/forms/fields/#built-in-field-classes) , щоб переглянути більше цих полів.

Ці поля форми представляються користувачеві у вигляді [**віджета HTML**](https://docs.djangoproject.com/en/4.1/ref/forms/widgets/) (віджети відображають елементи html-форми), але якщо ми явно не вкажемо віджет для нашого поля форми, Django буде використовувати віджети за замовчуванням. Перевизначаючи стандартний віджет для поля форми, ми можемо кастомізувати наші поля форми.

У класі Meta ми можемо створити зв'язок між полями нашої моделі та різними полями, які ми хочемо мати у нашій формі, причому порядок перелічення полів має значення.

**Обробник маршруту**

Для додавання тегів у нашому застосунку необхідно відкрити файл noteapp/views.py та додати функцію обробник tag, яка відображатиме шаблон tag.html та обробляти POST запити на створення нових тегів.

**notes/noteapp/views.py**

def tag(request):

if request.method == 'POST':

form = TagForm(request.POST)

if form.is\_valid():

form.save()

return redirect(to='noteapp:main')

else:

return render(request, 'noteapp/tag.html', {'form': form})

return render(request, 'noteapp/tag.html', {'form': TagForm()})

Логіка роботи досить проста. Для GET запиту ми просто виконуємо рендер шаблону:

return render(request, 'noteapp/tag.html', {'form': TagForm()})

Для POST запиту ми виконуємо обробку форми form = TagForm(request.POST) та перевіряємо валідність введених даних форми функцією form.is\_valid(). Якщо дані валідні, то виконуємо збереження даних форми у базі даних form.save() і робимо перехід на головну сторінку return redirect(to='main'). При не валідних даних виконуємо повторний рендер шаблону return render(request, 'noteapp/tag.html', {'form': form}). Саме тут спрацює виведення помилки в інструкції {{ form.errors.name }} шаблону.

Залишилося додати всередину файлу noteapp/urls.py маршрут з обробником:

from django.urls import path

from . import views

app\_name = 'noteapp'

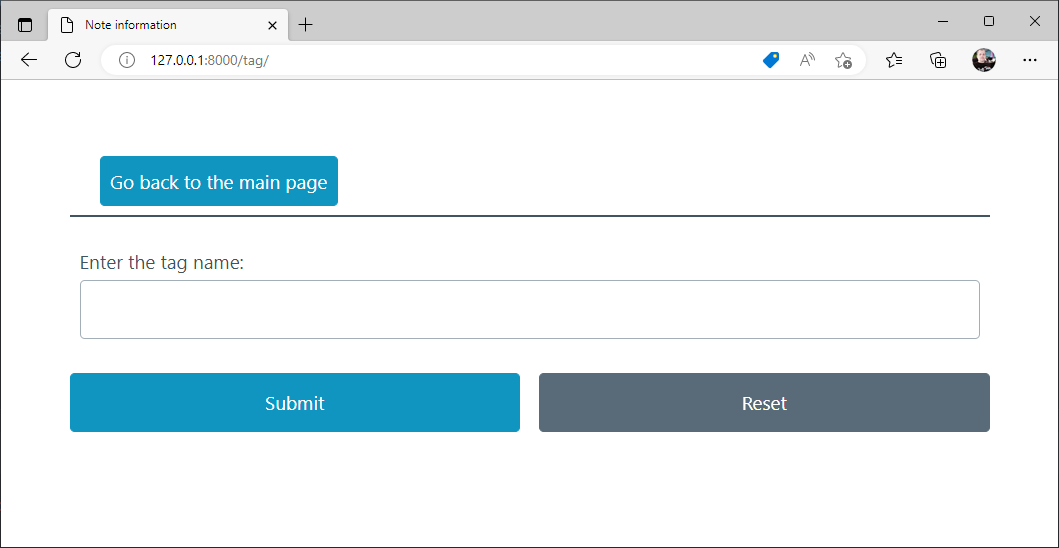
urlpatterns = [

path('', views.main, name='main'),

path('tag/', views.tag, name='tag'),

]

Тут за допомогою параметра name='tag' ми вказуємо ім'я маршруту. Це необхідно, щоб працювала зв'язка атрибуту action="{% url 'noteapp:tag' %}" форми з ім'ям маршруту. Змінна app\_name = 'noteapp' якраз визначає префікс для маршруту в атрибуті action.



Тепер ми можемо додавати будь-яку кількість тегів у нашому застосунку.

**Додавання нотаток**

Шаблон для додавання нотатки виглядатиме так.

**notes/noteapp/templates/noteapp/note.html**

{% extends "noteapp/base.html" %}

{% block content %}

<form method="POST" action="{% url 'noteapp:note' %}">

{% csrf\_token %}

<div style="padding: 10px">

<label> Note name:

{{ form.name }}

</label>

<span>{{ form.errors.name }}</span>

</div>

<div style="padding: 10px">

<label> Note description:

{{ form.description }}

</label>

<span>{{ form.errors.description }}</span>

</div>

<div style="padding: 10px">

<label> Select tags:

<select name="tags" multiple="multiple" required>

{% for tag in tags %}

<option value="{{tag.name}}">{{tag.name}}</option>

{% endfor %}

</select>

</label>

</div>

<div class="grid">

<button type="submit">Submit</button>

<button type="reset" class="secondary">Reset</button>

</div>

</form>

{% endblock %}

Тут повторюється той самий підхід, що й у шаблоні для тегів. Новим є множинний вибір назв тегів за допомогою тегу select, під час створення нотатки. Ми даємо користувачеві можливість додати кілька тегів за один раз.

Також створюємо форму для нотаток всередині файлу forms.py.

class NoteForm(ModelForm):

name = CharField(min\_length=5, max\_length=50, required=True, widget=TextInput())

description = CharField(min\_length=10, max\_length=150, required=True, widget=TextInput())

class Meta:

model = Note

fields = ['name', 'description']

exclude = ['tags']  
  
Ми виключили з перевірки форми теги exclude = ['tags'], оскільки це зв'язок багато-до-багатьох і ми його будемо обробляти в особливий спосіб.

Додамо функцію представлення нотатки у файл noteapp/views.py:

def note(request):

tags = Tag.objects.all()

if request.method == 'POST':

form = NoteForm(request.POST)

if form.is\_valid():

new\_note = form.save()

choice\_tags = Tag.objects.filter(name\_\_in=request.POST.getlist('tags'))

for tag in choice\_tags.iterator():

new\_note.tags.add(tag)

return redirect(to='noteapp:main')

else:

return render(request, 'noteapp/note.html', {"tags": tags, 'form': form})

return render(request, 'noteapp/note.html', {"tags": tags, 'form': NoteForm()})

Спочатку ми шукаємо всі теги tags=Tag.objects.all() — їх назви ми передаємо у шаблон для виведення в тезі select. Якщо тип запиту GET, то повертаємо шаблон для створення нотатки, попередньо прокинувши всередину всі існуючі теги tags — return render(request, 'noteapp/note.html', {"tags": tags, 'form': NoteForm()}).

Якщо метод запиту POST, збираємо дані з форми form=NoteForm(request.POST) та перевіряємо їх на валідність. При не валідних даних ми рендеримо шаблон із повідомленнями про помилки. Створюємо нотатку note і зберігаємо результат в базу даних new\_note = form.save(), але поки що у нас не прив'язані теги.

Отримуємо список тегів tags із запиту до БД choice\_tags=Tag.objects.filter(name\_\_in=request.POST.getlist('tags')). Тут варто пояснити кілька моментів. Перше, щоб отримати саме список з елемента форми, ми повинні використовувати метод getlist — request.POST.getlist('tags'). Друге - нам необхідно використовувати SQL оператор IN для перевірки входження тегу в отриманий список. Django використовує підхід вказівки імені поля name, символу подвійного підкреслення і сам оператор — filter(name\_\_in=request.POST.getlist('tags')). Для кращого розуміння ознайомтеся з [**документацією**](https://docs.djangoproject.com/en/4.1/topics/db/examples/many_to_many/)**.**

Додаємо теги до нотатки у циклі за допомогою методу add. Перенаправляємо користувача на головну сторінку.

Якщо ж метод запиту GET, то повертаємо шаблон для створення нотатки, попередньо прокинувши всередину всі існуючі теги tags.

Всередині файлу noteapp/urls.py додамо маршрут обробки нотаток:

from django.urls import path

from . import views

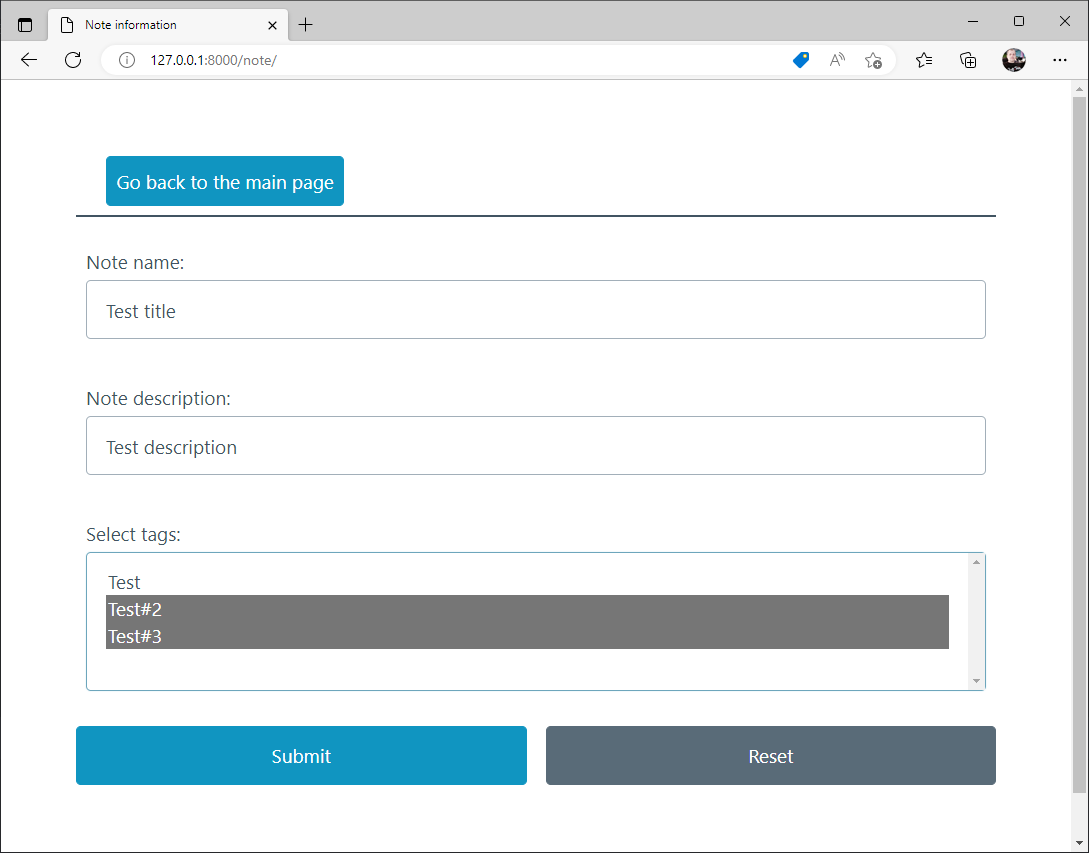
urlpatterns = [

path('', views.main, name='main'),

path('note/', views.note, name='note'),

path('tag/', views.tag, name='tag'),

]



Процес створення нотатки завершено, переходимо до сторінки відображення створеної нотатки.

****Сторінка відображення нотатки****

Для відображення нашої нотатки ми будемо використовувати маршрут http://127.0.0.1:8000/detail/<int:note\_id>, де <int:note\_id> - це id, унікальний ідентифікатор нотатки в базі даних.

Додамо функцію detail для виведення шаблону нотатки у файл noteapp/views.py:

from django.shortcuts import render, redirect, get\_object\_or\_404

...

def detail(request, note\_id):

note = get\_object\_or\_404(Note, pk=note\_id)

return render(request, 'noteapp/detail.html', {"note": note})

Вона досить проста, із нового - функція get\_object\_or\_404, яка знаходить необхідну нотатку з бази даних або повертає відповідь 404, якщо такої нотатки з note\_id не існує.

Всередині файлу noteapp/urls.py додамо новий маршрут відображення вмісту нотатки:

from django.urls import path

from . import views

app\_name = 'noteapp'

urlpatterns = [

path('', views.main, name='main'),

path('tag/', views.tag, name='tag'),

path('note/', views.note, name='note'),

path('detail/<int:note\_id>', views.detail, name='detail'),

]

Тут в описі маршруту ми вказуємо параметр note\_id для передачі id нотатки всередину функції представлення detail.

Шаблон відображення нотатки має наступний вигляд:

**notes/noteapp/templates/noteapp/detail.html**

{% extends "noteapp/base.html" %} {% load extract\_tags %} {% block content %}

<h1>{{note.name}}</h1>

<p>Description: <b>{{note.description}}</b></p>

<p>Time of creation: <kbd>{{note.created}}</kbd></p>

{% if note.done %}

<p>

Status:

<ins>Completed</ins>

</p>

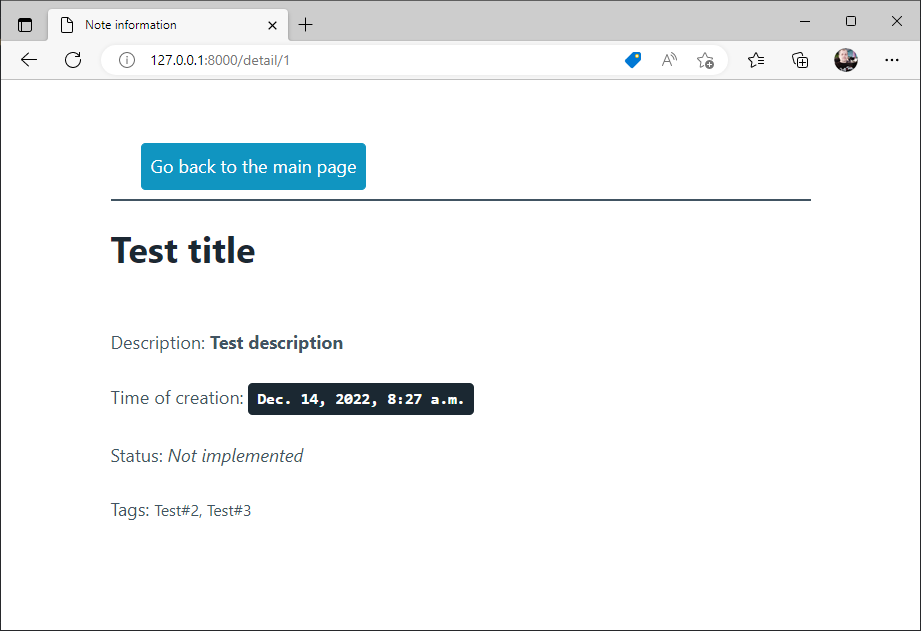
{% else %}

<p>Status: <i>Not implemented</i></p>

{% endif %}

<p>Tags: <small> {{ note.tags|tags }} </small></p>

{% endblock %}



Тут все як звичайно для шаблонів, але потрібно пояснити, як ми отримуємо перелік тегів для нотатки. Нам, наприклад, хотілося б вивести всі теги нотатки через кому.

Відповідно до [**документації**](https://docs.djangoproject.com/en/4.0/topics/db/queries/#many-to-many-relationships), щоб отримати список тегів у нотатки, необхідно використовувати запит до БД note.tags.all(), але шаблони Django, на відміну від шаблонів Jinja2, забороняють виклик функцій всередині себе.

Допоможе нам [**механізм створення**](https://docs.djangoproject.com/en/dev/howto/custom-template-tags/#howto-custom-template-tags) власних тегів для шаблонів.

Для цього, всередині програми, потрібно створити папку templatetags і в ній файл extract\_tags.py, назва файлу довільна, з таким вмістом:

**notes/noteapp/templatetags/extract\_tags.py**

from django import template

register = template.Library()

def tags(note\_tags):

return ', '.join([str(name) for name in note\_tags.all()])

register.filter('tags', tags)

Необхідно підвантажити файл всередині шаблону {% load extract\_tags %} і зробити виведення тегів за допомогою передачі note.tags, як аргументу для спеціального тегу tags:

<p>Tags: <small> {{ note.tags|tags }} </small></p>

Переходимо до відображення списку наших нотаток на головній сторінці застосунку.

**Відображення списку нотаток**

Відображення списку нотаток будемо виконувати на головній сторінці index.html. Для цього спочатку змінимо представлення для функції main.

def main(request):

notes = Note.objects.all()

return render(request, 'noteapp/index.html', {"notes": notes})

Також у файлі представлень views.py додамо дві функції. Перша функція set\_done буде відповідати, що нотатка змінює статус на виконана:

def set\_done(request, note\_id):

Note.objects.filter(pk=note\_id).update(done=True)

return redirect(to='noteapp:main')

Друга функція delete\_note відповідає за видалення нотатки.

def delete\_note(request, note\_id):

Note.objects.get(pk=note\_id).delete()

return redirect(to='noteapp:main')

Додамо маршрути для виклику функцій у файлі noteapp/urls.py:

from django.urls import path

from . import views

app\_name = 'noteapp'

urlpatterns = [

path('', views.main, name='main'),

path('tag/', views.tag, name='tag'),

path('note/', views.note, name='note'),

path('detail/<int:note\_id>', views.detail, name='detail'),

path('done/<int:note\_id>', views.set\_done, name='set\_done'),

path('delete/<int:note\_id>', views.delete\_note, name='delete'),

]

Сам шаблон index.html виглядатиме так:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8"/>

<title>Organizer</title>

<link rel="stylesheet" href="https://unpkg.com/@picocss/pico@latest/css/pico.min.css"/>

</head>

<body>

<main class="container">

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

<ul>

<li><a href="{% url 'noteapp:note' %}" role="button">Add note</a></li>

<li><a href="{% url 'noteapp:tag' %}" role="button">Add Tag</a></li>

</ul>

</nav>

{% for note in notes %}

<ul>

<li>

{% if not note.done %}

<a href="{% url 'noteapp:detail' note.id %}" role="button" class="outline"> {{ note.name }} </a>

<a href="{% url 'noteapp:set\_done' note.id %}" role="button"> Mark Done </a>

<a href="{% url 'noteapp:delete' note.id %}" role="button" class="contrast"> Delete note </a>

{% else %}

<a href="{% url 'noteapp:detail' note.id %}" role="button" class="secondary"><s>{{ note.name }}</s></a>

{% endif %}

</li>

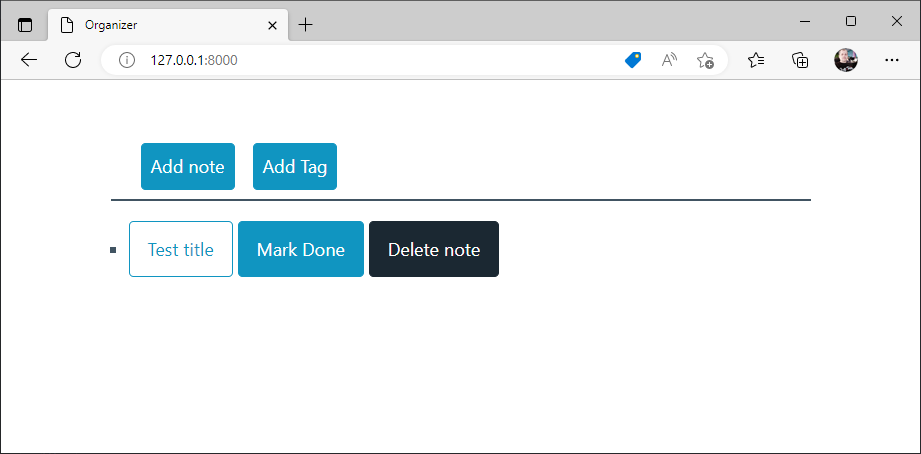
</ul>

{% endfor %}

</main>

</body>

</html>



Тут ми додали верхнє меню для переходу на сторінку тегів і нотаток:

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

<ul>

<li><a href="{% url 'noteapp:note' %}" role="button">Add note</a></li>

<li><a href="{% url 'noteapp:tag' %}" role="button">Add Tag</a></li>

</ul>

</nav>

Далі у циклі виводимо інформацію про існуючі нотатки.

{% for note in notes %}

<ul>

<li>

{% if not note.done %}

<a href="{% url 'noteapp:detail' note.id %}" role="button" class="outline"> {{ note.name }} </a>

<a href="{% url 'noteapp:set\_done' note.id %}" role="button"> Mark Done </a>

<a href="{% url 'noteapp:delete' note.id %}" role="button" class="contrast"> Delete note </a>

{% else %}

<a href="{% url 'noteapp:detail' note.id %}" role="button" class="secondary"><s>{{ note.name }}</s></a>

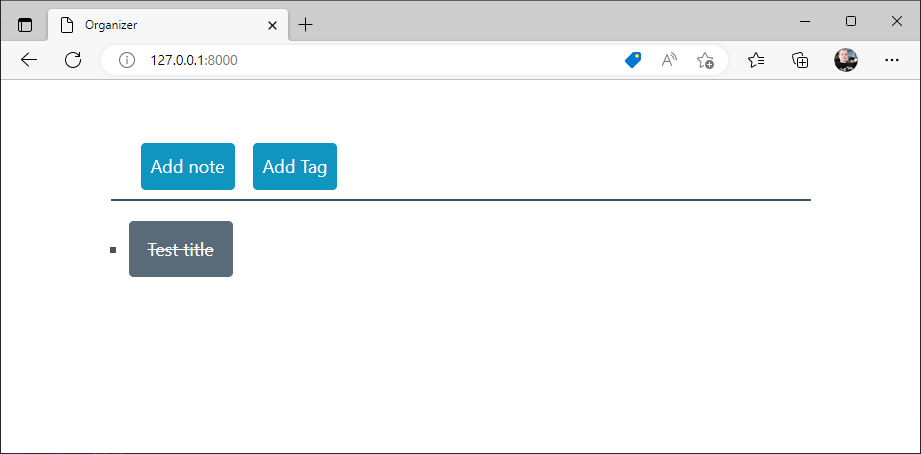
{% endif %}

</li>

</ul>

{% endfor %}

Тут ми розділяємо виведення нотатки, виконаної та не виконаної {% if not note.done %}. Помітимо нотатку виконаною та побачимо, що вигляд головної сторінки зміниться.

Також зверніть увагу, як передається параметр note\_id, наприклад, для маршруту detail/<int:note\_id>. Ми вказуємо {% url 'noteapp:detail' note.id %}, де говоримо який маршрут noteapp:detail нам потрібно вибрати для посилання, а далі через пробіл вказуємо значення note.id для параметра note\_id.

**Додаємо користувачів**

У нашого застосунку є великий недолік – всі нотатки та теги є спільними для будь-якого користувача нашого застосунку. У реальних застосунках такого немає. Нам необхідно додати користувачів. Для цього необхідно реалізувати механізм реєстрації нових користувачів та входу до нашого застосунку.

Створимо новий застосунок users.

python manage.py startapp users

Команда створить папку users в корені проекту. Потім в settings.py додаємо наш застосунок users у константу INSTALLED\_APPS:

INSTALLED\_APPS = [

...

"users",

]

У модулі notes/urls.py додамо маршрут на наш новий застосунок:

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path("admin/", admin.site.urls),

path('', include('noteapp.urls')),

path('users/', include('users.urls')),

]

Ми використовували в імені шляху префікс users, щоб уникнути зіткнень з іншими маршрутами застосунку – це хороша практика і завжди дотримуйтесь її.

**Реєстрація користувачів**

Django поставляється із вбудованою моделлю користувача, основними атрибутами якої є:

* username — ім'я користувача
* password — пароль користувача
* email — електронна пошта
* first\_name — ім'я користувача
* last\_name — прізвище користувача

Якщо нам знадобиться більше атрибутів, ніж пропонує вбудована модель, є спосіб зробити це, і ми трохи далі в профілі користувача додамо можливість завантаження аватару. Але зараз ми будемо використовувати модель користувача Django, як вона є.

Далі нам потрібна форма, яка буде відображатися користувачам, перевіряти дані та зберігати їх у базі даних. На щастя, нам не потрібно винаходити велосипед, тому що Django має вбудовану просту у використанні UserCreationForm.

Створимо модуль forms.py всередині застосунку users.

**notes/users/forms.py**

from django import forms

from django.contrib.auth.forms import UserCreationForm

from django.contrib.auth.models import User

class RegisterForm(UserCreationForm):

username = forms.CharField(max\_length=100,

required=True,

widget=forms.TextInput())

password1 = forms.CharField(max\_length=50,

required=True,

widget=forms.PasswordInput())

password2 = forms.CharField(max\_length=50,

required=True,

widget=forms.PasswordInput())

class Meta:

model = User

fields = ['username', 'password1', 'password2']

Із вбудованої моделі ми вибрали найнеобхідніший мінімум - ім'я користувача та пароль. Це той мінімум, який необхідний, щоб увійти у застосунок.

Перейдемо до *views.py* застосунку *users* і скористаємося тільки-но створеною нами формою. Створимо обробник маршруту *signupuser*:

**notes/users/views.py**

from django.shortcuts import render, redirect

from .forms import RegisterForm

def signupuser(request):

if request.user.is\_authenticated:

return redirect(to='noteapp:main')

if request.method == 'POST':

form = RegisterForm(request.POST)

if form.is\_valid():

form.save()

return redirect(to='noteapp:main')

else:

return render(request, 'users/signup.html', context={"form": form})

return render(request, 'users/signup.html', context={"form": RegisterForm()})

Всередині папки *users* створимо вкладену папку t*emplates/users*, де зберігатимемо шаблони нашого застосунку. Додамо шаблон *signup.html*:

{% extends "noteapp/base.html" %}

{% block content %}

<div>

<h2>Sign Up</h2>

</div>

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

{% csrf\_token %}

<div>

<label> Username:

{{ form.username }}

</label>

<span>{{ form.errors.username }}</span>

</div>

<br>

<div>

<label> Password:

{{ form.password1 }}

</label>

<span>{{ form.errors.password1 }}</span>

</div>

<div>

<label> Password confirmation:

{{ form.password2 }}

</label>

<span>{{ form.errors.password2 }}</span>

</div>

<div class="grid">

<button type="submit">Submit</button>

<button type="reset" class="secondary">Reset</button>

</div>

</form>

{% endblock %}

Останній штрих - створюємо файл *urls.py* всередині застосунку *users* і поміщаємо туди маршрут реєстрації користувача:

from django.urls import path

from . import views

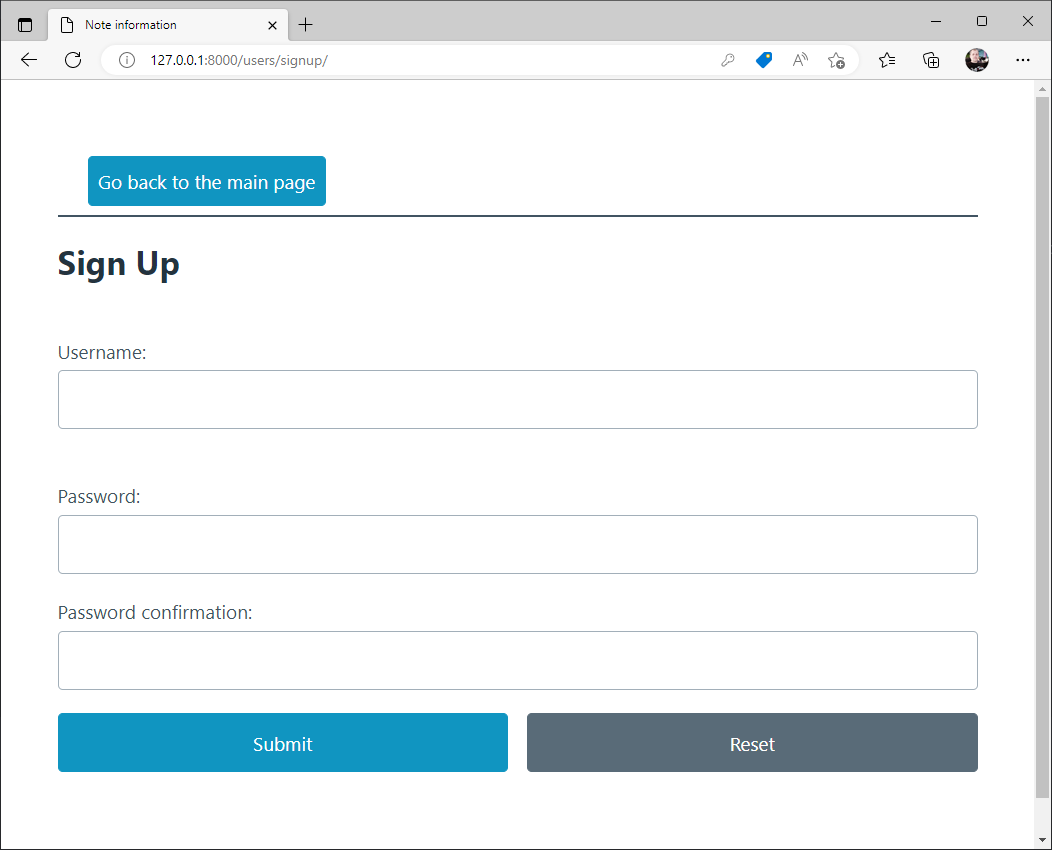
app\_name = 'users'

urlpatterns = [

path('signup/', views.signupuser, name='signup'),

]

Ми готові реєструвати користувачів у нашому застосунку



**Аутентифікація**

Після реєстрації користувач поміщається в базу даних. Нам потрібно провести аутентифікацію користувача. Аутентифікація стосується перевірки автентичності користувача: що він той, за кого себе видає. Для цього він повинен ввести ім'я користувача та пароль, який ми порівняємо з тим, що знаходиться у базі даних.

Форма для аутентифікації:

from django.contrib.auth.forms import UserCreationForm, AuthenticationForm

...

class LoginForm(AuthenticationForm):

class Meta:

model = User

fields = ['username', 'password']

Шаблон *login.html*:

{% extends "noteapp/base.html" %}

{% block content %}

{% if messages %}

<div>

{% for message in messages %}

<div {% if message.tags %} class="alert-{{ message.tags }}" {% endif %}>{{ message }}</div>

{% endfor %}

</div>

{% endif %}

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

{% csrf\_token %}

<div>

<label> Username:

{{ form.username }}

</label>

</div>

<div>

<label>Password:

{{ form.password }}

</label>

</div>

<div class="grid">

<button type="submit">Submit</button>

<button type="reset" class="secondary">Reset</button>

</div>

</form>

{% endblock %}

Наступний блок коду відповідає за виведення інформаційних повідомлень для користувачів:

{% if messages %}

<div>

{% for message in messages %}

<div {% if message.tags %} class="alert alert-{{ message.tags }}" {% endif %} role="alert">{{ message }}</div>

{% endfor %}

</div>

{% endif %}

Це одноразові повідомлення для користувача, які відображатимуться в наших шаблонах.

Перейдемо до *views.py* застосунку *users* і створимо обробник маршруту *loginuser*:

from django.contrib.auth import authenticate, login, logout

from django.contrib import messages

...

def loginuser(request):

if request.user.is\_authenticated:

return redirect(to='noteapp:main')

if request.method == 'POST':

user = authenticate(username=request.POST['username'], password=request.POST['password'])

if user is None:

messages.error(request, 'Username or password didn\'t match')

return redirect(to='users:login')

login(request, user)

return redirect(to='noteapp:main')

return render(request, 'users/login.html', context={"form": LoginForm()})

Тут для аутентифікації ми використовуємо функцію *authenticate*. Якщо користувач існує і пароль збігається, ми виконуємо логін *login(request, user)* користувача. В іншому випадку видаємо повідомлення 'Username or password didn\'t match' і повертаємося на сторінку аутентифікації.

Додаємо маршрут для входу користувачів у файл *urls.py*:

from django.urls import path

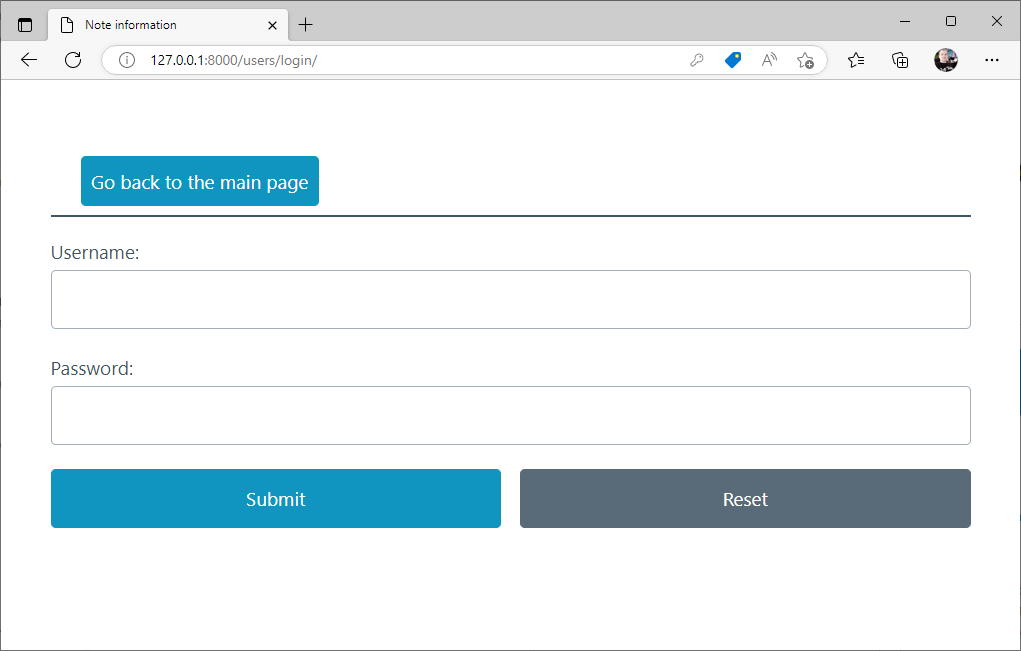
from . import views

app\_name = 'users'

urlpatterns = [

path('signup/', views.signupuser, name='signup'),

path('login/', views.loginuser, name='login'),

]

**Вихід користувача**

Бажано користувачеві давати можливість виходу із застосунку, не чекаючи завершення сеансу користувача. Для цього ми додамо маршрут */users/logout* та функцію обробник для нього.

**notes/users/views.py**

from django.contrib import messages

from django.contrib.auth import authenticate, login, logout

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

from django.shortcuts import render, redirect

from .forms import RegisterForm, LoginForm

...

@login\_required

def logoutuser(request):

logout(request)

return redirect(to='noteapp:main')

Декоратор *@login\_required* говорить про те, що виконання цієї функції можливе тільки для автентифікованого користувача.

Додаємо маршрут у файл *urls.py*:

from django.urls import path

from . import views

app\_name = 'users'

urlpatterns = [

path('signup/', views.signupuser, name='signup'),

path('login/', views.loginuser, name='login'),

path('logout/', views.logoutuser, name='logout'),

]

**Змінюємо головну сторінку**

Змінимо меню навігації у шаблоні головної сторінки, щоб ми могли потрапляти на сторінки реєстрації та логіну.

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

{% if user.is\_authenticated %}

<ul>

<li><a href="{% url 'noteapp:note' %}" role="button">Add note</a></li>

<li><a href="{% url 'noteapp:tag' %}" role="button">Add Tag</a></li>

</ul>

<ul>

<li>Welcome <b>{{ user.username }}!</b></li>

<li><a href="{% url 'users:logout' %}" role="button">Logout</a></li>

</ul>

{% else %}

<ul>

<li><a href="{% url 'users:signup' %}" role="button">Sign Up</a></li>

<li><a href="{% url 'users:login' %}" role="button">Login</a></li>

</ul>

{% endif %}

</nav>

Тепер у меню є привітання для користувача *<li>Welcome <b>{{user.username}}!</b></li>* та всі посилання для переходу на потрібні сторінки. Причому за допомогою інструкції *{% if user.is\_authenticated %}* ми змінюємо вигляд меню для користувача, що увійшов на сайт, і що не увійшов.

**З'єднуємо користувачів та нотатки**

Зараз сутності користувачів та нотаток жодним чином не пов'язані між собою. Виконаємо цей зв'язок.

**Зміна моделей**

Нам необхідно змінити моделі застосунку *noteapp*:

**notes/noteapp/models.py**

from django.db import models

from django.contrib.auth.models import User

# Create your models here.

class Tag(models.Model):

name = models.CharField(max\_length=25, null=False)

user = models.ForeignKey(User, on\_delete=models.CASCADE, default=1)

class Meta:

constraints = [

models.UniqueConstraint(fields=['user', 'name'], name='tag of username')

]

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

return f"{self.name}"

class Note(models.Model):

name = models.CharField(max\_length=50, null=False)

description = models.CharField(max\_length=150, null=False)

done = models.BooleanField(default=False)

created = models.DateTimeField(auto\_now\_add=True)

tags = models.ManyToManyField(Tag)

user = models.ForeignKey(User, on\_delete=models.CASCADE, default=1)

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

return f"{self.name}"

Зв'язуємо теги та нотатки відношенням один до багатьох user=models.ForeignKey(User, on\_delete=models.CASCADE, default=1). Оскільки у нас вже є створені теги та нотатки, вони повинні належати якомусь користувачеві. Ми, за замовчуванням, присвоюємо їх суперкористувачеві default=1, який створювався першим і має унікальний ключ 1.

Робимо теги унікальним складеним (constraint) ключем за двома полями user та name.

class Meta:

constraints = [

models.UniqueConstraint(fields=['user', 'name'], name='tag of username')

]

Створюємо та виконуємо міграції:

python manage.py makemigrations

Виведення:

Migrations for 'noteapp':

noteapp\migrations\0002\_note\_user\_tag\_user\_alter\_tag\_name\_and\_more.py

- Add field user to note

- Add field user to tag

- Alter field name on tag

- Create constraint tag of username on model tag

python manage.py migrate

Виведення:

Operations to perform:

Apply all migrations: admin, auth, contenttypes, noteapp, sessions

Running migrations:

Applying noteapp.0002\_note\_user\_tag\_user\_alter\_tag\_name\_and\_more... OK

**Зміна представлень**

Виведення головної сторінки

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

from django.shortcuts import render, redirect, get\_object\_or\_404

from .forms import TagForm, NoteForm

from .models import Tag, Note

def main(request):

notes = Note.objects.filter(user=request.user).all() if request.user.is\_authenticated else []

return render(request, 'noteapp/index.html', {"notes": notes})

Оскільки головна сторінка однакова для зареєстрованого користувача і незареєстрованого, то ми перевіряємо це і, щоб не було помилки при виведенні шаблону, створюємо порожній список для notes. У користувача, що не пройшов аутентифікацію, очевидно жодних notes бути не може.

Робота із тегами. Кожен обробник потрібно обгорнути декоратором *@login\_required*, тому що тільки зареєстровані користувачі можуть працювати з нотатками.

@login\_required

def tag(request):

if request.method == 'POST':

form = TagForm(request.POST)

if form.is\_valid():

tag = form.save(commit=False)

tag.user = request.user

tag.save()

return redirect(to='noteapp:main')

else:

return render(request, 'noteapp/tag.html', {'form': form})

return render(request, 'noteapp/tag.html', {'form': TagForm()})

Тепер нам потрібно додавати користувача у нотатку та теги. Тому процес створення тегу ускладнився.

tag = form.save(commit=False)

tag.user = request.user

tag.save()

Спочатку ми створюємо тег, але умовою *commit=False* не зберігаємо його одразу в базу даних. Додаємо користувача до тегу *tag.user=request.user* і тільки після цього зберігаємо в базу даних *tag.save()*.

Робота з нотатками

@login\_required

def note(request):

tags = Tag.objects.filter(user=request.user).all()

if request.method == 'POST':

form = NoteForm(request.POST)

if form.is\_valid():

new\_note = form.save(commit=False)

new\_note.user = request.user

new\_note.save()

choice\_tags = Tag.objects.filter(name\_\_in=request.POST.getlist('tags'), user=request.user)

for tag in choice\_tags.iterator():

new\_note.tags.add(tag)

return redirect(to='noteapp:main')

else:

return render(request, 'noteapp/note.html', {"tags": tags, 'form': form})

return render(request, 'noteapp/note.html', {"tags": tags, 'form': NoteForm()})

Шукаємо теги користувача *tags=Tag.objects.filter(user=request.user).all()*. Тепер ми скрізь при запиті додаємо умову *user=request.user*. Ситуація при збереженні нотатки така сама, як і при збереженні тегу.

Сторінка відображення нотатки

@login\_required

def detail(request, note\_id):

note = get\_object\_or\_404(Note, pk=note\_id, user=request.user)

return render(request, 'noteapp/detail.html', {"note": note})

Функції set\_done і delete\_note

@login\_required

def set\_done(request, note\_id):

Note.objects.filter(pk=note\_id, user=request.user).update(done=True)

return redirect(to='noteapp:main')

@login\_required

def delete\_note(request, note\_id):

Note.objects.get(pk=note\_id, user=request.user).delete()

return redirect(to='noteapp:main')

Тепер у кожного користувача свої нотатки та свої унікальні теги. Кожен користувач бачить лише те, що йому належить. Нам залишилося лише зробити профіль користувача.

**Профіль користувача**

Тепер ми збираємося створити профіль для користувача, включивши додаткову інформацію, таку як фотографія профілю або іншими словами – аватар.

Насамперед, давайте створимо представлення профілю всередині *views.py*

from django.shortcuts import render

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

@login\_required

def profile(request):

return render(request, 'users/profile.html')

Ми змінимо це представлення пізніше, щоб користувачі могли оновлювати свій профіль та завантажувати аватар. Декоратор *@login\_required* обмежує доступ для користувачів, що не увійшли в систему, а значить сторонній користувач не зможе отримати доступ до сторінки профілю.

Відкрийте програму *users* і у файлі *urls.py* додайте маршрут для перегляду профілю.

**notes/users/urls.py**

from django.urls import path

from . import views

app\_name = 'users'

urlpatterns = [

path('signup/', views.signupuser, name='signup'),

path('login/', views.loginuser, name='login'),

path('logout/', views.logoutuser, name='logout'),

path('profile/', views.profile, name='profile'),

]

**Шаблон профілю**

Створіть наступний шаблон для представлення в каталозі шаблонів застосунку користувачів *notes/users/templates/users*.

**notes/users/templates/users/profile.html**

{% extends "noteapp/base.html" %}

{% block content %}

<div>

<h1>Profile page: {{user.username}}</h1>

<img src="{{ user.profile.avatar.url }}" alt="avatar of user"/>

</div>

{% if messages %}

<div>

{% for message in messages %}

<div {% if message.tags %} class="alert-{{ message.tags }}" {% endif %}>{{ message }}</div>

{% endfor %}

</div>

{% endif %}

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

{% csrf\_token %}

<div class="form-row">

<div class="col-md-5">

<div class="form-group">

<label class="small mb-1">Change Avatar:</label>

{{ profile\_form.avatar }}

<span>{{ profile\_form.errors.avatar }}</span>

</div>

</div>

</div>

<div class="grid">

<button type="submit">Save Changes</button>

<button type="reset" class="secondary">Reset</button>

</div>

</form>

{% endblock content %}

Головне, що тут у форми ми додали атрибут *enctype="multipart/form-data"*, щоб форма могла працювати із завантаженням файлів.

**Модель профілю**

Тепер необхідно виконати розширення користувальницької моделі за допомогою зв'язку один-до-одного. Потрібно змоделювати наш профіль таким чином, щоб у базі даних зберігалися зображення профілю користувача. Тому в шаблоні ми отримуємо доступ до аватару через користувача *user.profile.avatar.url*.

Коли ми хочемо зберегти додаткову інформацію про користувача і вона не пов'язана з аутентифікацією, ми можемо створити нову модель, яка має зв'язок один-до-одного з користувачем. У Django зв'язок один-до-одного між моделями створюється за допомогою поля моделі *OneToOneField*.

У відношеннях один-до-одного один запис у таблиці пов'язаний з одним і лише одним записом іншої таблиці за допомогою зовнішнього ключа. Екземпляр моделі користувача пов'язаний з одним і тільки одним екземпляром профілю.

**notes/users/models.py**

from django.db import models

from django.contrib.auth.models import User

from PIL import Image

# Extending User Model Using a One-To-One Link

class Profile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

avatar = models.ImageField(default='default\_avatar.jpeg', upload\_to='profile\_images')

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

return self.user.username

# resizing images

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

super().save()

img = Image.open(self.avatar.path)

if img.height > 250 or img.width > 250:

new\_img = (250, 250)

img.thumbnail(new\_img)

img.save(self.avatar.path)

Перший аргумент *user=models.OneToOneField(User, on\_delete=models.CASCADE)* вказує, з якою моделлю буде пов'язана поточна модель, яка в нашому випадку є моделлю *User*. Другий аргумент *on\_delete=models.CASCADE* означає, що якщо користувач видалений, видаліть і його профіль.

Для поля аватара *avatar = models.ImageField(default='default\_avatar.jpeg', upload\_to='profile\_images')* першим аргументом є зображення за замовчуванням *default='default\_avatar.jpeg'*, яке використовується для користувача, якщо він не завантажує його самостійно. Другим аргументом є каталог, в який завантажуються зображення *upload\_to='profile\_images'*.

Користувачі можуть завантажувати фотографії для профілю будь-якого розміру, необхідно примусово обмежувати розмір зображення для зберігання з метою економії дискового простору. Бібліотека *pillow* є однією з найпоширеніших бібліотек для обробки зображень, яка дозволяє нам виконувати різні види маніпуляцій із зображеннями у python.

Django вимагає, щоб ми встановили цю бібліотеку щоразу, коли працюємо з полем *ImageField*, тому додамо цей пакет.

poetry add pillow

Саме обмеження зображення виконуємо у методі *save*.

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

super().save()

img = Image.open(self.avatar.path)

if img.height > 250 or img.width > 250:

new\_img = (250, 250)

img.thumbnail(new\_img)

img.save(self.avatar.path)

Ми перевизначаємо метод save для збереження даних у базі даних. Не забуваємо викликати метод батька *super().save()*. Дивимося, що якщо розмір зображення, що завантажується, перевищує 250 на 250 пікселів, то ми стискаємо його до необхідного нам розміру.

Ще одним важливим кроком є реєстрація моделі профілю всередині програми *admin.py*.

from django.contrib import admin

from .models import Profile

# Register your models here.

admin.site.register(Profile)

Щоб зміни набули чинності в нашій базі даних, давайте виконаємо міграцію.

python manage.py makemigrations

Migrations for 'users':

users\migrations\0001\_initial.py

- Create model Profile

python manage.py migrate

Operations to perform:

Apply all migrations: admin, auth, contenttypes, noteapp, sessions, users

Running migrations:

Applying users.0001\_initial... OK

**Робота з аватаром користувача**

Під час роботи з медіафайлами в Django ми повинні змінити деякі налаштування, щоб зберігати файли локально та обслуговувати їх за потреби.

Зокрема, нам потрібно встановити значення для констант MEDIA\_URL та MEDIA\_ROOT в налаштуваннях — файл settings.py:

MEDIA\_ROOT - повний шлях до каталогу, в якому зберігаються завантажені файли. Зазвичай ми зберігаємо такі файли, створюючи каталог всередині базового каталогу проекту.

MEDIA\_URL - це базова URL-адреса для обслуговування файлів мультимедіа. Це те, що дозволяє нам отримувати доступ до медіа через наш веб-браузер.

**notes/notes/settings.py**

...

MEDIA\_ROOT = BASE\_DIR / 'media'

MEDIA\_URL = '/media/'

...

Після цього необхідно налаштувати *urls.py* проекту для обслуговування завантажених користувачем медіафайлів під час розробки (коли *debug=True*).

**notes/notes/urls.py**

from django.conf import settings

from django.conf.urls.static import static

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path("admin/", admin.site.urls),

path('', include('noteapp.urls')),

path('users/', include('users.urls')),

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

Тепер ми можемо додавати файли в кореневу папку мультимедіа, і Django буде обслуговувати їх з URL-адреси носія.

Створіть каталог media всередині кореня проекту та помістіть туди будь-яке зображення, яке буде зображенням профілю за замовчуванням – notes/media/default\_avatar.jpeg.

**Сигнали в Джанго**

Зараз всі зареєстровані користувачі не мають свого профілю. Ми повинні перейти на сторінку адміністратора, щоб створити профіль, як вже існуючим користувачам, так і тим, які будуть створюватися. У цьому застосунку необхідно, щоб профіль створювався автоматично під час реєстрації нового користувача. Для цього в Django використовують сигнали — *Signals*.

Сигнали використовуються для виконання будь-якої дії щодо модифікації/створення конкретного запису в базі даних.

У сигналі маємо такі основні поняття.

*Відправник (Sender):* Sender, зазвичай, - це модель, яка повідомляє одержувача про події.

*Одержувач (Receiver):* Receiver, зазвичай, - це функція, яка працює з даними після того, як подія сталася, наприклад, коли екземпляр користувача ось-ось буде збережений в базі даних.

Зв'язок між відправниками та одержувачами здійснюється через «диспетчерів сигналів». Використовуючи сигнали, ми можемо створити екземпляр профілю безпосередньо під час створення нового екземпляра користувача всередині бази даних.

Django рекомендує нам розміщувати сигнали в каталозі застосунку як єдиний модуль. Тому створіть файл *signals.py* всередині застосунку користувача.

**notes/users/signals.py**

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

from django.contrib.auth.models import User

from django.dispatch import receiver

from .models import Profile

@receiver(post\_save, sender=User)

def create\_profile(sender, instance, created, \*\*kwargs):

if created:

Profile.objects.create(user=instance)

@receiver(post\_save, sender=User)

def save\_profile(sender, instance, \*\*kwargs):

instance.profile.save()

Тут *create\_profile* - це функція приймача, яка запускається щоразу при створенні користувача. Користувач є відправником, який несе відповідальність за направлення повідомлення. Параметр *post\_save*, декоратора *@receiver* - сигнал, що посилається в кінці методу збереження.

Наведений вище код робить те, що після того, як метод *save* моделі *User* завершив виконання, він посилає сигнал *post\_save* до функції приймача *create\_profile* і ми створюємо профіль користувача. Потім функція *save\_profile* отримає сигнал для збереження екземпляра профілю цього користувача в базі даних.

Наступним кроком є підключення приймачів *ready* у файл *apps.py* шляхом імпорту модуля сигналів.

**notes/users/apps.py**

from django.apps import AppConfig

class UsersConfig(AppConfig):

default\_auto\_field = "django.db.models.BigAutoField"

name = "users"

def ready(self):

import users.signals # noqa

Ми перевизначили метод *ready* конфігурації застосунку *users* для виконання завдання ініціалізації, яка реєструє сигнали.

**Форма профілю**

Створимо форму, де користувачі оновлюватимуть свій профіль і відображатимуть інформацію всередині шаблону.

Для створення форм ми будемо використовувати клас forms.ModelForm, який дозволяє створювати форми, що взаємодіють з певною моделлю всередині бази даних.

**notes/users/forms.py**

from .models import Profile

...

class ProfileForm(forms.ModelForm):

avatar = forms.ImageField(widget=forms.FileInput())

class Meta:

model = Profile

fields = ['avatar']

Тепер ProfileForm взаємодіє з моделлю профілю та дозволяє користувачам оновлювати свій профіль.

Тепер оновимо представлення, щоб додати щойно створені форми.

**notes/users/views.py**

from .forms import RegisterForm, LoginForm, ProfileForm

...

@login\_required

def profile(request):

if request.method == 'POST':

profile\_form = ProfileForm(request.POST, request.FILES, instance=request.user.profile)

if profile\_form.is\_valid():

profile\_form.save()

messages.success(request, 'Your profile is updated successfully')

return redirect(to='users:profile')

profile\_form = ProfileForm(instance=request.user.profile)

return render(request, 'users/profile.html', {'profile\_form': profile\_form})

Коли форма відправлена, нам потрібно передати дані запиту *request.POST* у форму *ProfileForm*. Але для форми профілю є дані у вигляді файлу зображення, які надійшли разом із запитом. Ці дані файлу розміщені в об'єкті *request.FILES*, тому їх також потрібно.

Користувальницька форма очікує на екземпляр користувача, оскільки вона працює з моделлю користувача *User*, тому ми для форми профілю передаємо в екземпляр моделі профілю *instance=request.user.profile*.

Останній штрих - додамо в меню на головній сторінці перехід на сторінку профілю.

<nav style="margin-bottom: 20px; border-bottom: 2px solid; padding-left: 30px">

{% if user.is\_authenticated %}

<ul>

<li><a href="{% url 'noteapp:note' %}" role="button">Add note</a></li>

<li><a href="{% url 'noteapp:tag' %}" role="button">Add Tag</a></li>

</ul>

<ul>

<li>Welcome <b>{{ user.username }}!</b></li>

<li><a href="{% url 'users:profile' %}" role="button">Profile</a></li>

<li><a href="{% url 'users:logout' %}" role="button">Logout</a></li>

</ul>

{% else %}

<ul>

<li><a href="{% url 'users:signup' %}" role="button">Sign Up</a></li>

<li><a href="{% url 'users:login' %}" role="button">Login</a></li>

</ul>

{% endif %}

</nav>

**NOTE**

*Пам'ятайте, що для вже існуючих користувачів профіль користувача потрібно створити через адмін-панель. Для нових профіль буде створюватися автоматично.*

**Механізм скидання пароля**

Іншим частим використанням надсилання листів є операція скидання пароля, якщо користувач його забув. Цей сценарій ми розглянемо на прикладі фреймворку Django для проекту з модуля **"Модуль 10: Основи роботи з Django".**

Оскільки Django має готові представлення, які дають змогу реалізувати механізм скидання пароля для користувача, якщо він його забуде. Подивімось, як ми можемо додати цю функціональність до нашого застосунку.

**Робочий процес**

Спочатку розберемо процес відновлення пароля:

1. Користувач запитує скидання пароля, вказуючи свою адресу електронної пошти у формі скидання пароля.
2. Django генерує унікальний токен скидання пароля і надсилає лист із посиланням на скидання пароля на адресу електронної пошти користувача. Посилання вмикає маркер скидання пароля як параметр.
3. Коли користувач натискає на посилання скидання пароля, Django перевіряє маркер скидання пароля, і з’являється форма для введення нового пароля.
4. Користувач вводить новий пароль та надсилає форму.
5. Django оновлює пароль користувача та повідомляє про це користувача.

**Налаштовуємо параметри електронної пошти**

Щоб надсилати листи у Django, ми повинні додати всередину файлу налаштувань settings.py нашого застосунку наступні константи:

**notes/notes/settings.py**

EMAIL\_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

EMAIL\_HOST = 'smtp.meta.ua'

EMAIL\_PORT = 465

EMAIL\_STARTTLS = False

EMAIL\_USE\_SSL = True

EMAIL\_USE\_TLS = False

EMAIL\_HOST\_USER = 'example@meta.ua'

EMAIL\_HOST\_PASSWORD = 'secretPassword'

DEFAULT\_FROM\_EMAIL = EMAIL\_HOST\_USER

Розберімо, що означають ці параметри:

* EMAIL\_BACKEND — backends, який буде використовуватися для надсилання електронної пошти. У цьому випадку використовується backends простого протоколу передачі пошти SMTP: django.core.mail.backends.smtp.EmailBackend;
* EMAIL\_HOST — ім’я хоста SMTP-сервера, який буде використовуватися для надсилання електронної пошти. У нашому випадку встановлено значення smtp.meta.ua;
* EMAIL\_PORT — порт, який буде використовуватися у разі підключення до SMTP-сервера зі встановленим значенням 465;
* EMAIL\_STARTTLS — прапорець, який вказує, чи використовувати безпеку транспортного рівня TLS у разі підключення до SMTP-сервера. Ми будемо використовувати протокол SSL і ставимо значення False;
* EMAIL\_USE\_SSL — прапорець, який вказує, чи варто використовувати SSL у разі підключення до SMTP-сервера. У нашому випадку ми встановлюємо значення True;
* EMAIL\_USE\_TLS — прапорець, який вказує, чи варто використовувати TLS у разі підключення до SMTP-сервера;
* EMAIL\_HOST\_USER — ім’я користувача, що використовується під час аутентифікації на SMTP-сервері;
* EMAIL\_HOST\_PASSWORD — пароль, який буде використовуватися під час аутентифікації на SMTP-сервері;
* DEFAULT\_FROM\_EMAIL — адреса електронної пошти за замовчуванням, яка буде використовуватися як адреса "from" для всіх листів, що надсилаються застосунком. Ми встановлюємо значення EMAIL\_HOST\_USER.

**Під час реєстрації додаємо поле email**[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-13/module-13-01/django-email#%D0%BF%D1%96%D0%B4-%D1%87%D0%B0%D1%81-%D1%80%D0%B5%D1%94%D1%81%D1%82%D1%80%D0%B0%D1%86%D1%96%D1%97-%D0%B4%D0%BE%D0%B4%D0%B0%D1%94%D0%BC%D0%BE-%D0%BF%D0%BE%D0%BB%D0%B5-email)

Під час реєстрації нового користувача необхідно додати поле email:

**notes/users/templates/users/signup.html**

{% extends "noteapp/base.html" %}

{% block content %}

<div>

<h2>Sign Up</h2>

</div>

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

{% csrf\_token %}

<div>

<label> Username:

{{ form.username }}

</label>

<span>{{ form.errors.username }}</span>

</div>

<div>

<label> Email:

{{ form.email }}

</label>

<span>{{ form.errors.email }}</span>

</div>

<div>

<label> Password:

{{ form.password1 }}

</label>

<span>{{ form.errors.password1 }}</span>

</div>

<div>

<label> Password confirmation:

{{ form.password2 }}

</label>

<span>{{ form.errors.password2 }}</span>

</div>

<div class="grid">

<button type="submit">Submit</button>

<button type="reset" class="secondary">Reset</button>

</div>

</form>

{% endblock %}

Та змінити клас RegisterForm:

**notes/users/forms.py**

from django import forms

from django.contrib.auth.forms import UserCreationForm, AuthenticationForm

from django.contrib.auth.models import User

from .models import Profile

class RegisterForm(UserCreationForm):

username = forms.CharField(max\_length=100,

required=True,

widget=forms.TextInput())

email = forms.CharField(max\_length=100,

required=True,

widget=forms.TextInput())

password1 = forms.CharField(max\_length=50,

required=True,

widget=forms.PasswordInput())

password2 = forms.CharField(max\_length=50,

required=True,

widget=forms.PasswordInput())

class Meta:

model = User

fields = ['username', 'email', 'password1', 'password2']

class LoginForm(AuthenticationForm):

class Meta:

model = User

fields = ['username', 'password']

class ProfileForm(forms.ModelForm):

avatar = forms.ImageField(widget=forms.FileInput())

class Meta:

model = Profile

fields = ['avatar']

**Додаємо маршрути**

Додамо наступні маршрути, щоб реалізувати механізм скидання пароля.

**notes/users/urls.py**

from django.contrib.auth.views import PasswordResetDoneView, PasswordResetConfirmView, PasswordResetCompleteView

from django.urls import path

from . import views

app\_name = 'users'

urlpatterns = [

path('signup/', views.signupuser, name='signup'),

path('login/', views.loginuser, name='login'),

path('logout/', views.logoutuser, name='logout'),

path('profile/', views.profile, name='profile'),

path('reset-password/', views.ResetPasswordView.as\_view(), name='password\_reset'),

path('reset-password/done/', PasswordResetDoneView.as\_view(template\_name='users/password\_reset\_done.html'),

name='password\_reset\_done'),

path('reset-password/confirm/<uidb64>/<token>/',

PasswordResetConfirmView.as\_view(template\_name='users/password\_reset\_confirm.html',

success\_url='/users/reset-password/complete/'),

name='password\_reset\_confirm'),

path('reset-password/complete/',

PasswordResetCompleteView.as\_view(template\_name='users/password\_reset\_complete.html'),

name='password\_reset\_complete'),

]

Наприклад, користувач намагається залогінитися, але забув пароль. Користувач натискає на посилання «Forgot Password?» у шаблоні login.html

**notes/users/templates/users/login.htm**l

{% extends "noteapp/base.html" %}

{% block content %}

{% if messages %}

<div>

{% for message in messages %}

<div {% if message.tags %} class="alert-{{ message.tags }}" {% endif %}>{{ message }}</div>

{% endfor %}

</div>

{% endif %}

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

{% csrf\_token %}

<div>

<label> Username:

{{ form.username }}

</label>

</div>

<div>

<label>Password:

{{ form.password }}

</label>

</div>

<div class="grid">

<button type="submit">Submit</button>

<button type="reset" class="secondary">Reset</button>

</div>

<div class="small text-center">

<a href="{% url "users:password\_reset" %}" class="text-center"><i>Forgot Password?</i></a>

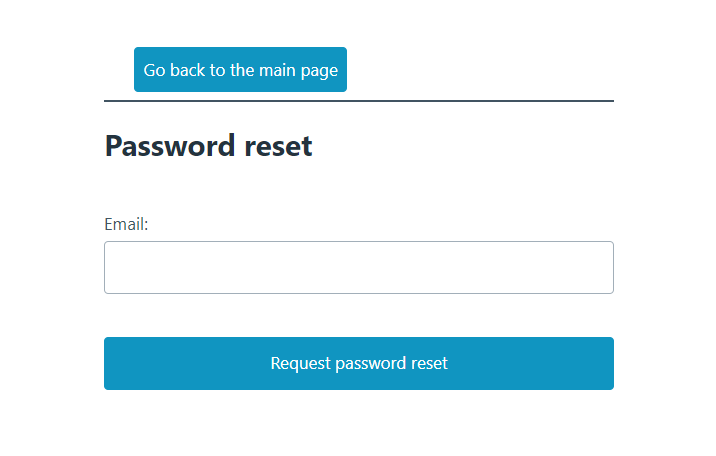
</div>

</form>

{% endblock %}

**Представлення для скидання пароля**

Це приведе його до маршруту reset-password/ зі сторінкою, де йому буде запропоновано ввести свою адресу електронної пошти. Це представлення ми будемо обробляти шляхом класу ResetPasswordView. Це представлення демонструє форму, в якій користувач надсилає свою адресу електронної пошти.



Django перевіряє, існує чи ні користувач із наданою адресою електронної пошти в базі даних. Якщо користувач існує, то генерує посилання для скидання пароля, яке буде використовуватися лише один раз. Після цього він надсилає це посилання на адресу електронної пошти користувача.

Клас ResetPasswordView є класом-представленням для скидання пароля користувача Django. Він наслідується від PasswordResetView з django.contrib.auth.views та SuccessMessageMixin з django.contrib.messages.views.

**notes/users/views.py**

from django.contrib import messages

from django.contrib.auth import authenticate, login, logout

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

from django.contrib.auth.views import PasswordResetView

from django.contrib.messages.views import SuccessMessageMixin

from django.shortcuts import render, redirect

from django.urls import reverse\_lazy

...

class ResetPasswordView(SuccessMessageMixin, PasswordResetView):

template\_name = 'users/password\_reset.html'

email\_template\_name = 'users/password\_reset\_email.html'

html\_email\_template\_name = 'users/password\_reset\_email.html'

success\_url = reverse\_lazy('users:password\_reset\_done')

success\_message = "An email with instructions to reset your password has been sent to %(email)s."

subject\_template\_name = 'users/password\_reset\_subject.txt'

Клас SuccessMessageMixin надає можливість показувати повідомлення про успішне виконання дії в користувацькому інтерфейсі. Клас PasswordResetView є класом-представленням, який вже реалізує функціональність скидання пароля.

Наше представлення ResetPasswordView встановлює кілька атрибутів для налаштування процесу скидання пароля:

* template\_name — ім’я шаблону, який буде використовуватися для сторінки скидання пароля;
* email\_template\_name — ім’я шаблону, який буде використовуватися для листа про скидання пароля;
* html\_email\_template\_name — ім’я шаблону, який буде використовуватися для HTML-версії листа про скидання пароля;
* success\_url — URL-адреса для перенаправлення після успішного надсилання листа про скидання пароля;
* success\_message — повідомлення, яке буде показано користувачеві після успішного надсилання листа про скидання пароля;
* subject\_template\_name — ім’я шаблону, який буде використовуватися для теми листа про скидання пароля;

Визначимо шаблон для сторінки скидання пароля.

**notes/users/templates/users/password\_reset.html**

{% extends "noteapp/base.html" %}

{% block content %}

<h2>Password reset</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Request password reset</button>

</form>

{% endblock %}

Визначимо шаблон для HTML-версії листа про скидання пароля

**notes/users/templates/users/password\_reset\_email.html**

{% autoescape off %}

<body>

<p>Hi {{ user.username }},</p>

<p>You requested a password reset for your account. Please follow this link to set a new password:</p>

<p>

<a href="{{ protocol }}://{{ domain }}{% url 'users:password\_reset\_confirm' uidb64=uid token=token %}">

{{ protocol }}://{{ domain }}{% url 'users:password\_reset\_confirm' uidb64=uid token=token %}

</a>

</p>

<p>If you didn't request a password reset, please ignore this email.</p>

<p>Thanks,</p>

<p>The Our Team</p>

{% endautoescape %}

Лист містить кілька абзаців тексту, які надають користувачеві інформацію про процес скидання пароля. Посилання на сторінку скидання пароля генерується за допомогою тегу шаблону {% url %} та шаблону URL users:password\_reset\_confirm. Ми також використовуємо наступні змінні в шаблоні:

* user.username — ім’я користувача, який запросив скидання пароля;
* protocol — протокол (наприклад, "http" або "https"), який використовується в URL-адресі сторінки скидання пароля;
* domain — домен вебсайту;
* uid — закодоване в base64 представлення ідентифікатора користувача;
* token — токен скидання пароля, який був згенерований для користувача;

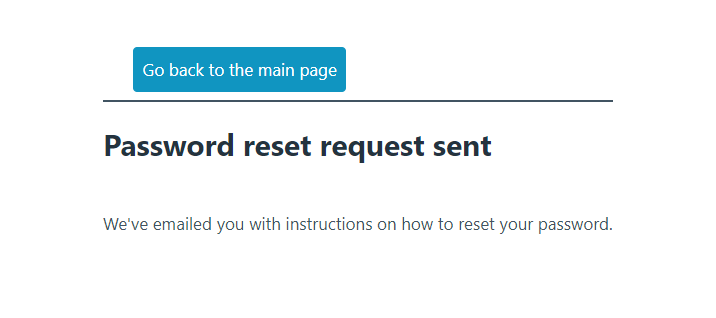
Шаблон для теми листа – це простий текстовий файл:

**notes/users/templates/users/password\_reset\_subject.txt**

Password Reset

**Представлення успішного відправлення листа**

Після надсилання листа Django, відповідно до параметра success\_url = reverse\_lazy('users:password\_reset\_done') класу ResetPasswordView, виконає редірект на маршрут reset-password/done/. Цей маршрут обробляється представленням PasswordResetDoneView.as\_view(template\_name='users/password\_reset\_done.html').



Визначаємо шаблон password\_reset\_done.html:

**notes/users/templates/users/password\_reset\_done.html**

{% extends "noteapp/base.html" %}

{% block content %}

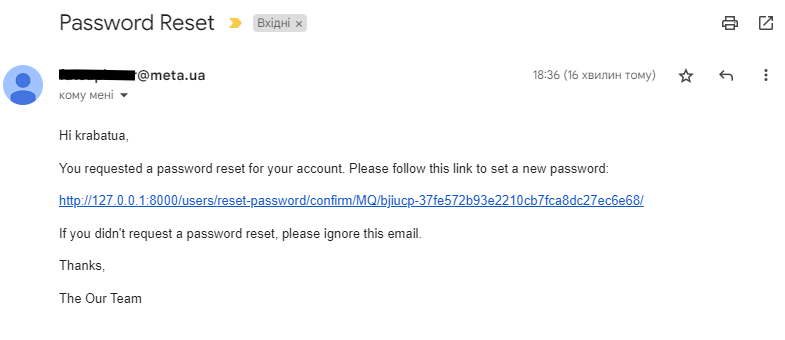
<h2>Password reset request sent</h2>

<p>We've emailed you with instructions on how to reset your password.</p>

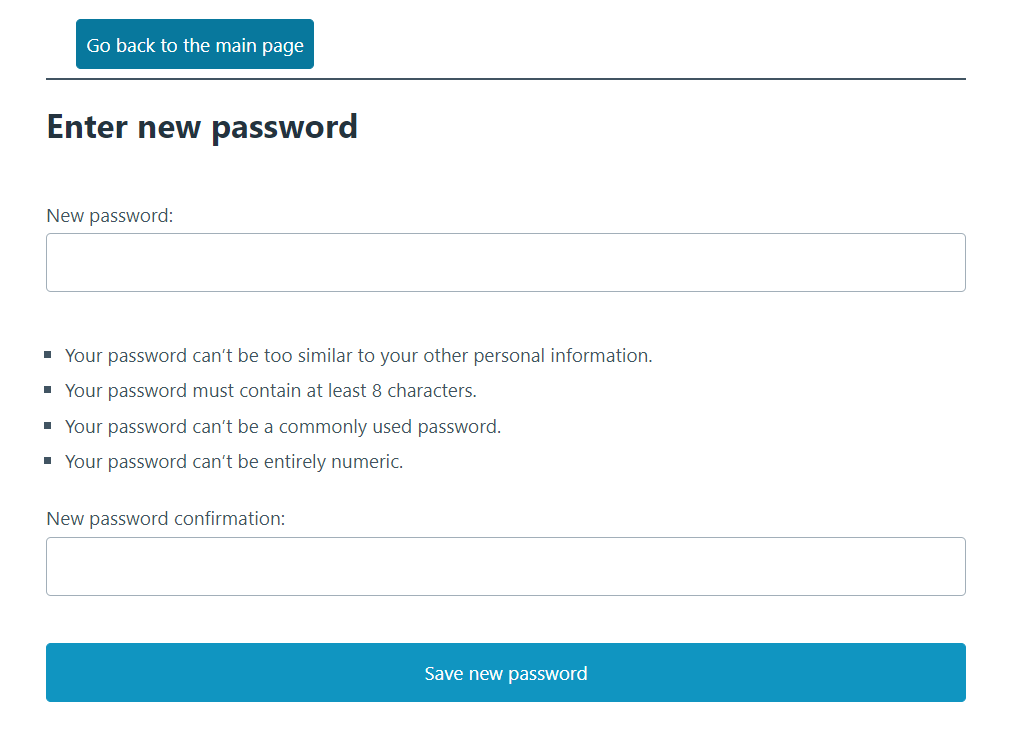
{% endblock %}

**Скидання пароля**

Необхідно зайти до своєї поштової скриньки та знайти лист з інструкцією скидання пароля.



Після натискання на вказане посилання ми перейдемо на маршрут /reset-password/confirm/<uidb64>/<token>/. Де ми повинні створити новий пароль.



За обробку цієї форми відповідає представлення PasswordResetConfirmView. Представлення буде використовувати шаблон users/password\_reset\_confirm.html для відображення форми підтвердження скидання пароля і, після успішного підтвердження, перенаправлятиме користувача на URL /users/reset-password/complete/.

Метод as\_view використовується для перетворення представлення на основі класу на об’єкт, що викликається, який ми використовуємо в конфігурації маршруту.

**notes/users/templates/users/password\_reset\_confirm.html**

{% extends "noteapp/base.html" %}

{% block content %}

<h2>Enter new password</h2>

<form method="post">

{% csrf\_token %}

{{ form.as\_p }}

<button type="submit">Save new password</button>

</form>

{% endblock %}

**Представлення успішного підтвердження**

Після того, як ми введемо новий пароль і відправимо його на збереження в базу даних, Django повідомить користувача про це за допомогою класу PasswordResetCompleteView.

...

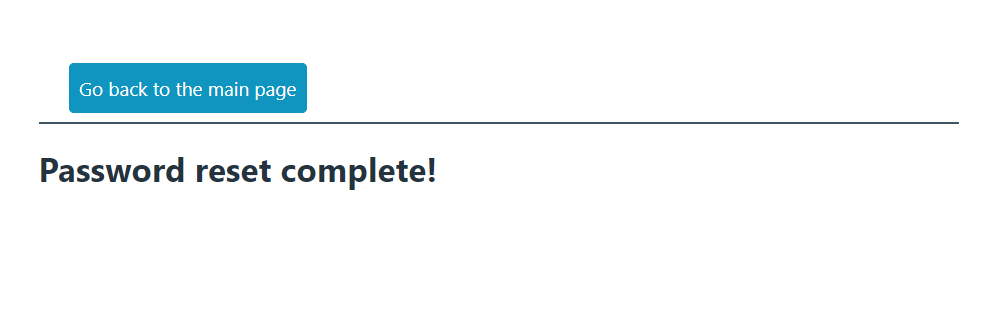
path('reset-password/complete/', PasswordResetCompleteView.as\_view(template\_name='users/password\_reset\_complete.html'),

name='password\_reset\_complete'),

...

Цей код створює шаблон URL для сторінки завершення скидання пароля.

Шаблон прив’язується до класу PasswordResetCompleteView із вбудованих представлень аутентифікації Django і використовує шаблон users/password\_reset\_complete.html для відображення сторінки завершення.



Ось так виглядає шаблон:

**notes/users/templates/users/password\_reset\_complete.html**

{% extends "noteapp/base.html" %}

{% block content %}

<h2>Password reset complete!</h2>

{% endblock %}

**Змінні середовища Django**

У проекті Django всі конфіденційні дані захисту знаходяться у файлі notes/notes/settings.py.

Ми будемо використовувати пакет[**Django Environ**](https://django-environ.readthedocs.io/en/latest/install.html)**.** Встановімо його:

poetry add django-environ

Після цього імпортуємо environ у settings.py та ініціалізуємо його:

import environ

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

env = environ.Env()

environ.Env.read\_env(BASE\_DIR / '.env')

Створимо файл .env у кореневому каталозі та не забудемо додати його до .gitignore. Оголосимо змінні середовища в .env і переконаємося, що ми не використовуємо лапки навколо рядків.

**.env**

SECRET\_KEY=django-insecure-6njcs!&=f6o)\_=tqieoj6+wzljxwg=71l9th&n\_m8w$o1c\*xe3

DATABASE\_NAME=django\_email

DATABASE\_USER=postgres

DATABASE\_PASSWORD=567234

DATABASE\_HOST=127.0.0.1

DATABASE\_PORT=5432

EMAIL\_HOST=smtp.meta.ua

EMAIL\_PORT=465

EMAIL\_HOST\_USER=example@meta.ua

EMAIL\_HOST\_PASSWORD=secretPassword

Замінимо всі посилання на змінні середовища у файлі settings.py

1. Секретний ключ:

SECRET\_KEY = env('SECRET\_KEY')

2.Підключення до бази даних

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql\_psycopg2',

'NAME': env('DATABASE\_NAME'),

'USER': env('DATABASE\_USER'),

'PASSWORD': env('DATABASE\_PASSWORD'),

'HOST': env('DATABASE\_HOST'),

'PORT': env('DATABASE\_PORT'),

}

}

3.Змінні для надсилання листів

EMAIL\_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

EMAIL\_HOST = env('EMAIL\_HOST')

EMAIL\_PORT = env('EMAIL\_PORT')

EMAIL\_STARTTLS = False

EMAIL\_USE\_SSL = True

EMAIL\_USE\_TLS = False

EMAIL\_HOST\_USER = env('EMAIL\_HOST\_USER')

EMAIL\_HOST\_PASSWORD = env('EMAIL\_HOST\_PASSWORD')

DEFAULT\_FROM\_EMAIL = EMAIL\_HOST\_USER

Все, ми забезпечили безпеку всіх наших конфіденційних даних у проекті Django.