Geekbrains

Разработка веб-платформы для персонального хранения данных с системой оценки и изоляцией пользователей на **Python** и библиотеки **Django** для оптимизации управления записями**.**

Программа:

Python-разработчик. Специалист

Калинин Егор Леонидович

Ижевск

2025

Введение

**Темой проекта** является создание сайта с использованием Python и библиотеки Django для оптимизации управления и возможности оценки хранимых записей текстовых данных, фотографий и др. .

**Цель:** Создание сайта с возможностью хранения текстовых/фото данных возможностью их оценки на языке Python и библиотеки Django, а так же возможностью группировать по связанным темам и простоты последующего поиска.

* **Актуальность проблемы:** Каждый пользователь хранит текстовые/фото данные в разных источниках и поэтому их поиск бывает затруднителен, так же ввиду отсутствия “рейтинга” у данных нет возможности определять её ценность для конкретного пользователя, сайт же позволяет сохранить, сгруппировать по категориям, а так же связанным темам, дать/изменить индивидуальную оценку личным данным.

**Задачи:**

* Изучить литературу, касающуюся темы исследования, для определения современных тенденций и практик.
* Выявить ключевые потребности пользователей для успешного использования системы.
* Разработать и протестировать продукт.
* Подготовить систему для запуска на сервере для дальнейшей эксплуатации.

**Инструменты:** PyCharm 2024.2.5, pyllow 11.1.0, Python 3.10, Django 4.2.1, Git,

Django-extensions 3.2.3, django-debug-toolbar 4.2.0, unidecode 1.3.8.

**Состав команды:** Калинин Егор Леонидович

**Теоретическая часть**

*1.* ***Введение***

Веб-приложение, разработанное в рамках данного дипломного проекта, представляет собой индивидуальное хранилище для пользователей, которое позволяет хранить записи, ссылки и изображения, а также задавать рейтинг своим данным. Система также обеспечивает изоляцию пользователей друг от друга, что позволяет каждому пользователю иметь личное пространство для хранения данных. Проект включает в себя важные аспекты безопасности, удобства использования и функциональности, что делает его универсальным инструментом для организации личных записей.

*2.* ***Обзор существующих решений***

Современные сервисы для хранения данных, такие как **Evernote**, **Google Keep** и **Pinterest**, предлагают функции для хранения заметок, ссылок и изображений. Эти платформы предоставляют пользователю возможность добавлять и организовывать записи с помощью тегов и категорий. Однако большинство из них не предоставляет изоляции пользователей или полного контроля над сохранёнными данными. Разработанное приложение на основе Django предоставляет дополнительную гибкость, включая индивидуальное хранилище для каждого пользователя с функцией рейтинга и приватности данных.

***3. Технологический стек***

Для реализации данного проекта была выбрана технология **Django** как веб-фреймворк для бэкенда, поскольку она предоставляет удобные средства для работы с различными базами данных, безопасности и аутентификации пользователей. В качестве базы данных использовалась **SQLite**, что идеально подходит для разработки локальных приложений и тестирования. Для фронтенда применялись стандартные технологии **HTML**, **CSS** и **JavaScript**. Дополнительно для обработки запросов и обновления данных использовалась библиотека **AJAX**.

***4. Моделирование данных***

В проекте была использована реляционная база данных, которая включает в себя несколько основных моделей: User, Category, Record, Rating. Каждая модель имеет соответствующие поля для хранения данных:

* **User** — модель, представляющая пользователей системы, с полями для имени, электронной почты и пароля.
* **Category** — модель для хранения категорий, в которых могут быть размещены записи пользователей. Эта модель имеет связи с таблицей **Record** через внешний ключ.
* **Record** — модель для хранения записей, содержащих ссылки, текстовые заметки и изображения. Каждая запись привязана к определённому пользователю и категории.
* **TagPost**— модель для хранения рейтинга каждой записи, который может быть выставлен пользователем.

Для обеспечения изоляции данных пользователей каждая запись привязывается к пользователю с помощью внешнего ключа, что исключает доступ других пользователей к личным данным.

***5. Аутентификация и изоляция пользователей***

Для защиты данных пользователей была реализована система аутентификации с использованием стандартных инструментов Django. Каждому пользователю предоставляется индивидуальная зона для хранения данных, и доступ к этим данным ограничен только самим пользователем. Система использует механизм сессий для управления состоянием пользователя в системе. При аутентификации пользователя его уникальный идентификатор сохраняется в сессии, что позволяет пользователю обращаться только к собственным данным и не иметь доступа к записям других пользователей.

***6. Рейтинг записей и взаимодействие с данными***

Для каждой записи в системе реализована система рейтинга, где пользователи могут оценивать свои записи с помощью простого интерфейса. Рейтинг вычисляется на основе голосов других пользователей, и каждый пользователь может выставить только одну оценку для записи. Для предотвращения фальсификации рейтингов был применён механизм контроля, который проверяет корректность оценок.

***7. Безопасность данных и защита от атак***

Важным аспектом разработки является безопасность данных пользователей. В проекте применены следующие меры:

* **Хеширование паролей** с использованием алгоритма **PBKDF2** для защиты паролей пользователей.
* **Защита от SQL-инъекций** благодаря использованию ORM Django, которая автоматически экранирует все параметры запросов.
* **Защита от CSRF и XSS атак** с помощью стандартных механизмов Django, которые предотвращают внедрение опасных скриптов и защищают от подделки запросов.

Также предусмотрена возможность восстановления паролей и двухфакторная аутентификация для повышения безопасности.

***8. Пользовательский интерфейс и взаимодействие с системой***

Пользовательский интерфейс приложения разработан с учётом принципов удобства и простоты. Главная страница представляет собой список записей пользователя, где отображаются названия, категории и рейтинги. Каждому пользователю предоставляется возможность добавить новые записи, изменять существующие и удалять их. Также на страницах предусмотрены кнопки для выставления оценок и для фильтрации записей по категориям. Интерфейс адаптивен, что позволяет использовать приложение как на мобильных, так и на десктопных устройствах.

***9. Заключение***

В рамках дипломного проекта был разработан веб-сайт для хранения записей, ссылок и изображений, который включает систему рейтинга и изоляции пользователей. Система предоставляет удобный интерфейс для работы с личными данными, обеспечивая безопасность и удобство использования. Реализованный проект продемонстрировал важность создания персонализированных хранилищ для пользователей и успешное применение современных технологий для решения поставленных задач.

**Общая структура приложения**

| manage.py - *отвечает за управление проектом*

|

+---siterecords – *ключевая директория проекта*

| | asgi.py - *используется для налаживания связи между приложением и веб-сервером*

| | settings.py - *содержит в себе все настройки проекта*

| | urls.py - *задаёт ассоциации url адресов с представлениями*

| | wsgi.py - *используется для налаживания связи между приложением и веб-сервером*

| | \_\_init\_\_.py

|

+---records – *папка с приложением, отвечающим за основное взаимодействие*

*пользователя и сервера*

| | admin.py - *здесь регистрируются модели, для административной части сайта Django*

| | apps.py - *используется для создания конфигурации приложения (имя, инициализация, и др.)*

| | models.py - *файл, который содержит модели базы данных проекта*

| | tests.py – *файл для тестов приложения*

| | urls.py - *для маршрутизации URL-адресов к соответствующим представлениям (views)*

| | views.py - *отвечает за обработку запросов и возврат ответов (логика приложения)*

| | \_\_init\_\_.py

| |

| +---migrations – *папка для файлов миграций (изменения в структуре БД)*

| | | 0001\_initial.py

| | | 0002\_alter\_category\_name\_alter\_records\_cat.py | | | 0003\_tagpost\_author.py | | | 0004\_alter\_records\_options\_records\_rating\_and\_more.py | | | 0005\_alter\_category\_author\_alter\_records\_author\_and\_more.py | | | \_\_init\_\_.py | +---templates – *папка с шаблонами HTML страниц*

+---media

| +---users *папка для загружаемых пользователями на сервер фото*

|

+---static - *содержит статические файлы приложения*

| +---admin

| | +---css

| | |

| | +---img

| | |

| | +---js

| |

| +---templatetags | \_\_init\_\_.py

| records\_tags.py

|

+---templates

| +---admin

| | | base\_site.html

| | base.html

|

+---users – *папка с приложением, отвечающим за аунтефикацию/авторизацию пользователей*

| | admin.py

| | apps.py

| | forms.py

| | models.py

| | tests.py

| | urls.py

| | views.py

| | \_\_init\_\_.py

| | authentication.py – *модуль, отвечающий за возможность авторизации через е-майл*

*| |* context\_processors.py *- шаблонный контекстный менеджер*

| +---migrations

| | | \_\_init\_\_.py

| |

| +---templates

| | \---users

| | login.html

| | register.html

| | password\_change\_form.htm | | profile.html | | register\_done.html

**Используемые библиотеки**

1. asgiref==3.7.2: Утилиты для работы с ASGI (Asynchronous Server Gateway Interface) в Python.

2. unidecode==1.3.8: Библиотека для определения и нормализации кодировки текста.

3. Django==4.2.1: веб-фреймворк для создания сложных веб-приложений на Python.

4. django-debug-toolbar==4.2.0: инструмент для отладки, который предоставляет полезную информацию о работе Django-приложения в процессе разработки. Он отображается в браузере и помогает разработчикам эффективно отслеживать производительность, запросы, ошибки и другие аспекты работы приложения.

5. ipython==8.14.0: мощная интерактивная оболочка для Python, которая значительно улучшает процесс работы с Python в консоли, предлагая удобные функции и возможности, которые делают разработку более эффективной и удобной.

6. python-dotenv==0.12.0: Позволяет загружать переменные окружения из .env файлов, для настройки конфигураций.

7. pillow==11.1.0: предоставляет удобный интерфейс для работы с изображениями, и его возможности включают создание, открытие, редактирование, манипуляцию и сохранение различных форматов изображений.

8. sqlparse==0.5.0: Простая библиотека для разбора SQL-запросов в удобочитаемый формат.

**Описание ключевой директории проекта**

Файлы asgi.py и wsgi.py создаются автоматически при создании проекта и не требуют изменений.

**asgi.py** - Этот файл используется для настройки приложения с ASGI-сервером, что позволяет обрабатывать веб-сокеты и асинхронные запросы.

**wsgi.py** - Этот файл настраивает WSGI-сервер для запуска вашего приложения, поддерживая стандартные HTTP-запросы.

Файл **\_\_init\_\_.py** - Указывает Python, что данная директория является пакетом. Он может быть пустым или содержать код инициализации(далее про него не будет упоминаться, так как выполняет ту же функцию).

./siterecords/settings.py

Файл **settings.py** - Содержит основные настройки проекта, такие как базы данных, файлы статик, параметры безопасности и другие конфигурации.

Разберем файл более подробно:

**import** environ

**from** pathlib **import** Path

# Переменная, которая указывает на корневую директорию проекта Django, используется для построения путей к другим папкам и файлам в проекте.

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

# Загрузка переменные окружения из файла .env. Это обычно используется для хранения конфиденциальных данных, таких как ключи API или настройки базы данных

env **=** environ.Env()

**environ.Env.read\_env()**

# ПРЕДУПРЕЖДЕНИЕ ПО технике БЕЗОПАСНОСТИ: храните секретный ключ, используемый при запуске, в секрете! Значение переменной SECRET\_KEY берется из переменных окружения операционной системы (из файла .env)

SECRET\_KEY **=** *env(*"SECRET\_KEY")

# DEBUG используется в настройках веб-приложений на Django для включения режима отладки

DEBUG **=** **False**

# Это список разрешенных хостов для приложения. В продакшн-режиме здесь указываются домены и IP-адреса, с которых приложение сможет принимать запросы

ALLOWED\_HOSTS **=** **[‘127.0.0.1’]**

INTERNAL\_IPS **=** **[‘127.0.0.1’]**

# Настройка указывает, куда перенаправлять пользователей после успешного входа в систему. В данном случае, после логина пользователь будет перенаправлен на главную страницу ("home")

LOGIN\_REDIRECT\_URL **=** "home"

# Настройка указывает, куда будет перенаправлен пользователь как он выйдет из своей учётной записи, в нашем случае будет перенаправлен на главную страницу ("home")

LOGOUT\_REDIRECT\_URL **=** "home"

# Настройка, которая указывает на URL на который будет перенаправлен пользователь, если он пытается получить доступ к защищенному ресурсу не будучи аутентифицированным.

LOGIN\_URL **=** "user:login"

# Список всех приложений, которые используются в проекте

INSTALLED\_APPS **=** **[**

"django.contrib.admin"**,**

"django.contrib.auth"**,**

"django.contrib.contenttypes"**,**

"django.contrib.sessions"**,**

"django.contrib.messages"**,**

"django.contrib.staticfiles"**,**

"django\_extensions"**,**

"records.apps.RecordsConfig"**,**

* "users.apps.UsersConfig"**,**

"debug\_toolbar"**,**

**]**

# Посредники (middleware) обрабатывают запросы и ответы на разных стадиях. Они могут выполнять такие действия, как обработка сессий, аутентификация, и защита от CSRF-атак

MIDDLEWARE **=** **[**

"django.middleware.security.SecurityMiddleware"**,**

"django.contrib.sessions.middleware.SessionMiddleware"**,**

"django.middleware.common.CommonMiddleware"**,**

"django.middleware.csrf.CsrfViewMiddleware"**,**

"django.contrib.auth.middleware.AuthenticationMiddleware"**,**

"django.contrib.messages.middleware.MessageMiddleware"**,**

"django.middleware.clickjacking.XFrameOptionsMiddleware"**,**

"django\_toolbar.middleware.DebugToolbarMiddleware"**,**

**]**

# Указывает, какой файл будет использоваться для маршрутизации (URL routing). В данном случае это siterecords.urls, где определяются все маршруты (URL) приложения

ROOT\_URLCONF **=** "siterecords.urls"

# Определяет настройки для обработки шаблонов. Указаны директории, где можно найти шаблоны, и необходимые контекстные процессоры, которые позволяют использовать разные переменные в шаблонах

TEMPLATES **=** **[**

**{**

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

"DIRS"**:** **[**

BASE\_DIR **/** "templates"**,**

**],**

"APP\_DIRS"**:** **True,**

"OPTIONS"**:** **{**

"context\_processors"**:** **[**

"django.template.context\_processors.debug"**,**

"django.template.context\_processors.request"**,**

"django.contrib.auth.context\_processors.auth"**,**

"django.contrib.messages.context\_processors.messages"**,**

"users.context\_processors.get\_records\_context"**,**

**],**

**},**

**},**

**]**

WSGI\_APPLICATION **=** "siterecords.wsgi.application"

# Определяем настройки для базы данных. "ENGINE" - Указывает, какой тип базы данных используется (в данном случае SQLite). "NAME" - Путь к файлу базы данных в корне проекта. Вначале пытаемся получить данные из .env, по умолчанию используются базовые настройки расположения и версия БД.

DATABASES **=** **{**

"default"**:** **{**

"ENGINE"**:** env("DB\_ENGINE ", default="django.db.backends.sqlite3")**,**

"NAME"**:** BASE\_DIR **/** env("DB\_NAME ", default="db.sqlite3")**,**

**}**

**}**

# список валидаторов паролей, которые используются для проверки надежности вводимых паролей

AUTH\_PASSWORD\_VALIDATORS **=** **[**

**{**

"NAME"**:** "django.contrib.auth.password\_validation.UserAttributeSimilarityValidator"**,**

**},**

**{**

"NAME"**:** "django.contrib.auth.password\_validation.MinimumLengthValidator"**,**

**},**

**{**

"NAME"**:** "django.contrib.auth.password\_validation.CommonPasswordValidator"**,**

**},**

**{**

"NAME"**:** "django.contrib.auth.password\_validation.NumericPasswordValidator"**,**

**},**

**]**

# Интернационализация. Устанавливает язык (русский) и часовой пояс (UTC)

LANGUAGE\_CODE **=** "ru-Ru"

TIME\_ZONE **=** "UTC"

USE\_I18N **=** **True**

USE\_TZ **=** **True**

# URL для статики (CSS, JS, изображения)

STATIC\_URL **=** "static/"

# Папка в которой временно хранятся данные и дополнительно во время разработки файлы будут искаться там. Так же дополним маршруты для медиа.

STATICFILES\_DIRS **=** [BASEDIR / "static",]

MEDIA\_URL **=** "/media/"

MEDIA\_ROOT **=** BASEDIR / "media"

# Папка, в которую будут собираться статические файлы для продакшн-окружения

STATIC\_ROOT **=** **(**"static"**,)**

# Определяет тип поля автоинкремента для первичных ключей, что помогает избежать проблем с ограничениями базы данных

# <https://docs.djangoproject.com/en/4.2/ref/settings/#default-auto-field>

DEFAULT\_AUTO\_FIELD **=** "django.db.models.BigAutoField"

# Модель пользователя для аутентификации, что позволяет использовать пользовательскую модель пользователя вместо стандартной

AUTHENTICATION\_BACKENDS **=** [

"django.contrib.auth.backends.ModelBackend"**,**

"users.authentication.EmailAuthBackend"**,**

]

Файл /siterecords/urls.py в Django отвечает за маршрутизацию (routing) URL-адресов приложения. Это urls.py ключевой директории.Рассмотрим код файла.

from django.contrib import admin  
from django.urls import path, include  
from django.http import HttpResponseNotFound  
  
urlpatterns = [  
 path('admin/', admin.site.urls),  
 path('', include('records.urls')),  
 path('users/', include('users.urls', namespace="users")),  
 path("\_\_debug\_\_/", include("debug\_toolbar.urls")),  
]  
  
handler404 = lambda request, exception: HttpResponseNotFound('<h1>Страница не найдена</h1>')  
  
admin.site.site\_header = "Панель администрирования"  
admin.site.index\_title = "Разделы для администрирования"

Определение маршрутов:

path('admin/', admin.site.urls): Этот маршрут устанавливает, что по адресу /admin/ будет доступен интерфейс администрирования.

Включение маршрутов приложений:

path('', include('records.urls')): Этот маршрут обрабатывает домашнюю страницу, перенаправляя на URL-ы, представления, определенные в приложении records.

path('users/', include('users.urls', namespace="users")): Этот маршрут отвечает за все URL, связанные с работой с пользователями, перенаправляя запросы на users.

Маршрут отладки:

path("\_\_debug\_\_/", include("debug\_toolbar.urls")): Этот маршрут добавляет пути для работы с панелью отладки, которую предоставляет Django Debug Toolbar. Этот инструмент полезен для отслеживания и анализа запросов на сервере.

Обработчик ошибок:

handler404 = lambda request, exception: HttpResponseNotFound('<h1>Страница не найдена</h1>'): Этот обработчик ошибок настраивает отображение страницы с сообщением "Страница не найдена" при возникновении ошибки 404 (когда запрашиваемая страница не найдена).

Настройка админки:

admin.site.site\_header = "Панель администрирования": Этот параметр изменяет заголовок страницы администрирования в интерфейсе.

admin.site.index\_title = "Разделы для администрирования": Этот параметр устанавливает название разделов на главной странице панели администрирования.

Этот файл маршрутизации связывает основные части приложения и добавляет кастомизацию, такую как обработчик ошибок и настройку панели администрирования, чтобы улучшить взаимодействие с приложением.

**Директория со статикой**

В папке «static» проекта Django хранятся статические файлы — изображения, JS и CSS-файлы.

Стилизация была реализована следующим образом:

**1. Основные стили для страницы**

Шрифт: Для всего документа используется шрифт Arial, что обеспечивает современный и чистый вид.

Отступы и размеры: Установлены нулевые отступы и поля (margin: 0; padding: 0;), чтобы исключить нежелательные расстояния между элементами. Высота и ширина установлены на 100% для того, чтобы элементы занимали весь экран.

**2. Ссылки (тег <a>)**

Ссылки стилизованы с основным цветом #2A6DB0 (синий) и без подчеркивания. При наведении цвет изменяется на #1b3649 появляется подчеркивание для улучшения визуального эффекта.

**3. Изображения (тег <img>)**

Для изображений установлена максимальная ширина 600px с автоматической высотой для сохранения пропорций.

Для изображений с классом img-article-left задается максимальная ширина 300px и отступы, чтобы они не перекрывали текст.

**4. Панель статьи (div.article-panel)**

Для панелей статей установлен светлый фон #eee и отступы, что помогает создать аккуратное разделение между блоками контента.

Параграфы с классами first и last используются для выравнивания текста влево и вправо.

**5. Меню**

Меню навигации (ul.mainmenu): Использован шрифт размером 20px и цвет #fdc073 для текста. Элементы меню расположены горизонтально с использованием флоатов.

При наведении цвет ссылок изменяется на #FDA83D.

Логотип в меню имеет фоновое изображение с фиксированными размерами.

**6. Блоки с кнопками (классы .btn и .button)**

Кнопки имеют круглые углы с радиусом 4px и цвет фона #2A6DB0 (синий), который меняется на более темный при наведении.

Кнопки также имеют белый текст и центрированы с помощью flexbox для расположения кнопок в строку с интервалами.

**7. Списки (ul)**

Для различных списков (.list-articles, .tags-list, .langitem и другие) используются стили, которые убирают стандартные маркеры списка и задают отступы между элементами.

В списке тегов добавлены отступы между элементами и используются различные цвета для активных элементов.

**8. Блоки контента**

Контент страницы и отдельных элементов, таких как статьи или категории, имеет отступы и выравнивание, чтобы текст не сливался с краями блоков.

В блоках с категорией текст сделан заглавными буквами, что помогает выделить ключевые элементы интерфейса.

**9. Пагинация и таблицы**

Для пагинации использованы элементы с круглыми кнопками, которые визуально выделяются за счет обводки и фоновых цветов, а также имеют эффект при наведении.

Для таблиц с контентом установлены отступы и максимальная ширина для улучшения восприятия данных.

**10. Подвал (footer)**

Для подвала страницы заданы темный фон #1b3649 и светлый текст #aaa, с небольшими отступами для аккуратного отображения информации.

**11. Форма ввода**

Введенные данные в формы имеют ограниченную ширину в 300px, что делает их удобными для заполнения.

Также предусмотрены стили для лейблов, ошибок и кнопок формы, чтобы создать четкое и понятное оформление для пользователей.

**12. Flexbox для выравнивания элементов**

Во многих местах используется flexbox, что позволяет выравнивать элементы по горизонтали и вертикали, а также делать интерфейс более гибким и адаптивным. Например, кнопки в #action-buttons выстраиваются в строку с равными промежутками.

**Директория с базовым шаблоном**

Базовый шаблон HTML в проекте Django — это основной файл, который используется для определения структуры и оформления страниц веб-приложения. Он обычно содержит общий HTML-код, такой как заголовок, меню навигации, подвал и ссылки на стили и скрипты. Затем другие шаблоны могут расширять этот базовый шаблон, добавляя свой контент.

{% load static %}  
{% load records\_tags %}  
  
<!DOCTYPE html>  
<html>  
<head>  
 <title>{{ title }}</title>  
 <link type="text/css" href="{% static 'records/css/styles.css' %}" rel="stylesheet" />  
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">  
 <link rel="shortcut icon" href="{% static 'records/images/icon.ico' %}" type="image/x-icon"/>  
 <meta name="viewport" content="width=device-width, initial-scale=1.0">  
</head>  
<body>  
<table class="table-page" border=0 cellpadding="0" cellspacing="0">  
<tr><td valign=top>  
{% block mainmenu %}  
 <div class="header">  
 <ul id="mainmenu" class="mainmenu">  
 <li class="logo"><a href="{% url 'home' %}"><div class="logo"></div></a></li>  
{% for m in mainmenu %}  
 <li><a href="{% url m.url\_name %}">{{m.title}}</a></li>  
{% endfor %}  
{% if user.is\_authenticated %}  
 <li class="last"><a href="{% url 'users:profile' %}">Профиль</a> | <a href="{% url 'users:logout' %}">Выйти</a></li>  
{% else %}  
 <li class="last"><a href="{% url 'users:login' %}">Войти</a> | <a href="{% url 'users:register' %}">Регистрация</a></li>  
{% endif %}  
 </ul>  
 <div class="clear"></div>  
 </div>  
{% endblock mainmenu %}  
  
  
<table class="table-content" border=0 cellpadding="0" cellspacing="0">  
<tr>  
 <td valign="top" class="left-chapters">  
 <ul id="leftchapters">  
 {% if cat\_selected == 0 %}  
 <li class="selected">Все категории</li>  
 {% else %}  
 <li><a href="{% url 'home' %}">Все категории</a></li>  
 {% endif %}  
 {% if user.is\_authenticated %}  
 {% show\_categories cat\_selected request %}  
 <li>{% show\_all\_tags %}</li>  
 {% else %}  
 <li>Здесь вы сможете увидеть вами созданные категории записей после авторизации/регистрации.</li>  
 {% endif %}  
  
 </ul>  
</td>  
<td valign="top" class="content">  
 {% block breadcrumbs %}  
 {% endblock %}  
 <div class="content-text">  
{% block content %}  
{% endblock %}  
 </div>  
  
{% block navigation %}  
{% endblock %}  
  
</td></tr></table>  
</td></tr>  
<tr><td valign=top>  
 <div id="footer">  
 <p>  
 &copy; 2024 Частичное или полное копирование информации с данного сайта для распространения на других ресурсах, в том числе и бумажных, строго запрещено.  
 <a href="/about/"> О сайте @</a>  
 <a href="/contacts/" style="float: right;">&nbsp;Контакты&nbsp;</a>  
 </p>  
</div>  
</td></tr></table>  
</body>  
</html>

Рассмотрим более подробно строки:

**{% load static %}** — это тег шаблона Django, который используется для загрузки статических файлов (CSS, JavaScript, изображения и т. д.) в шаблон. Когда мы используем этот тег, Django знает, что нужно искать статические файлы в каталоге, указанном в настройках проекта (STATICFILES\_DIRS, STATIC\_ROOT).

**{% load records\_tags %}** — в Django используется для загрузки пользовательских тегов шаблонов, которые определены в нашем проекте, чтобы затем использовать их в HTML-шаблонах. Теги шаблонов — это специальные функции, которые позволяют выполнять логические операции или получать данные, которые не могут быть выполнены обычными HTML-тегами.

**<title>{% block title %}{% endblock %}</title>** - Заголовок страницы, который будет заменяться в зависимости от контекста

В Django шаблонах конструкции {% block %} и {% endblock %} используются для определения областей, которые могут быть переопределены в дочерних шаблонах. Это часть механизма наследования шаблонов.

* Тег <body>:

В тегах <body> содержится основное содержимое веб-страницы.

В теге <header> (это верхняя часть сайта, расположенная выше блока с основным контентом и отображаемая на всех страницах)

- Содержит навигационное меню с кнопками для перехода на разные страницы, такие как "Главная", "Войти", "Регистрация", "Добавить запись", "Работа с категориями" и другие.

- Кнопки оформлены с использованием Bootstrap классов для красивого визуального отображения.

- Видимость кнопок зависит от того, аутентифицирован ли пользователь: если нет, доступна кнопка для входа; если да, показываются дополнительные кнопки, включая административные, если пользователь имеет соответствующие права.

В теге <main> (основной контент страницы)

- Содержит основной контент страницы, обернутый в контейнер.

- Использует {% block content %}{% endblock %} для динамической вставки содержимого из других шаблонов.

**Приложение records**

Приложение records, отвечает за обработку процедур.

Данное приложение производит валидацию и скачивание на сервер отправяемых пользователем данных, сохранение их в базу данных, формирование шаблонов главной страницы, страниц с формами и прочее.

Рассмотрим подробно код файлов приложения.

Файл apps.py

Apps.py — это модуль с настройками приложения в Django.

from django.apps import AppConfig  
  
  
class RecordsConfig(AppConfig):  
 verbose\_name = "Для работы с постами"  
 default\_auto\_field = 'django.db.models.BigAutoField'  
 name = 'records'  
  
class CategoriesConfig(AppConfig):  
 verbose\_name = "Категории"  
 default\_auto\_field = 'django.db.models.BigAutoField'  
 name = 'categories'

**Описание полей для RecordsConfig:**

- **name**: указывает имя приложения в Django. Здесь оно обозначено как "procedures\_app".

- **verbose\_name**: эта строка, которая задаёт читаемое название нашего приложения. Она используется для отображения человеческого имени приложения в административной панели Django и в других местах, где приложение отображается. В данном случае verbose\_name для приложения records будет отображать "Для работы с постами", что помогает пользователю понять, что это приложение занимается работой с записями (постами).

- **default\_auto\_field:** в данном случае, для всех моделей этого приложения будет использоваться тип поля BigAutoField, который создает автоинкрементируемое целочисленное поле с типом BIGINT (больший диапазон значений, чем у обычного AutoField).

Файл ./records/models.py

Модели в Django описывают структуру используемых данных.

Данные в программе хранятся в базах данных, и с помощью моделей осуществляется взаимодействие с базой данных.

from django.contrib.auth import get\_user\_model  
from django.core.validators import MinLengthValidator, MaxLengthValidator, MinValueValidator, MaxValueValidator  
from django.db import models  
from django.urls import reverse  
  
  
class PublishedManager(models.Manager):  
 def get\_queryset(self):  
 return super().get\_queryset().filter(is\_published=Records.Status.PUBLISHED)  
  
  
class Records(models.Model):  
 class Status(models.IntegerChoices):  
 PUBLISHED = 1, 'Активно'  
 DRAFT = 0, 'В корзине'  
  
 rating = models.IntegerField(  
 validators=[MinValueValidator(0), MaxValueValidator(10)], # Ограничение от 1 до 10  
 verbose\_name="Рэйтинг"  
 )  
 title = models.CharField(max\_length=255, verbose\_name="Заголовок")  
 slug = models.SlugField(max\_length=255, unique=True, db\_index=True, verbose\_name="Slug", validators=[  
 MinLengthValidator(5, message="Минимум 5 символов"),  
 MaxLengthValidator(100, message="Максимум 100 символов"),  
 ])  
 photo = models.ImageField(upload\_to="photos/%Y/%m/%d/", default=None,  
 blank=True, null=True, verbose\_name="Фото")  
 content = models.TextField(blank=True, verbose\_name="Текст статьи")  
 time\_create = models.DateTimeField(auto\_now\_add=True, verbose\_name="Время создания")  
 time\_update = models.DateTimeField(auto\_now=True, verbose\_name="Время последнего изменения")  
 is\_published = models.BooleanField(choices=tuple(map(lambda x: (bool(x[0]), x[1]), Status.choices)),  
 default=Status.PUBLISHED, verbose\_name="Статус")  
 cat = models.ForeignKey('Category', on\_delete=models.SET\_NULL, null=True, related\_name='posts',  
 verbose\_name="Категории")  
  
 tags = models.ManyToManyField('TagPost', blank=True, related\_name='tags', verbose\_name="Тэги")  
 author = models.ForeignKey(get\_user\_model(), on\_delete=models.SET\_NULL, null=True, related\_name='posts',  
 default=None)  
 objects = models.Manager()  
 published = PublishedManager()  
  
 def \_\_str\_\_(self):  
 return self.title  
  
 class Meta:  
 verbose\_name = "пост"  
 verbose\_name\_plural = "посты"  
 ordering = ['-rating', '-time\_create']  
 indexes = [  
 models.Index(fields=['-time\_create'])  
 ]  
  
 def get\_absolute\_url(self):  
 return reverse('post', kwargs={'post\_slug': self.slug})  
  
  
class Category(models.Model):  
 name = models.CharField(max\_length=100, db\_index=True, verbose\_name='Категория')  
 slug = models.SlugField(max\_length=255, unique=True, db\_index=True, verbose\_name='Slug')  
 author = models.ForeignKey(get\_user\_model(), on\_delete=models.SET\_NULL, null=True, related\_name='tag\_requests')  
  
 class Meta:  
 verbose\_name = "категория"  
 verbose\_name\_plural = "категории"  
  
 def \_\_str\_\_(self):  
 return self.name  
  
 def get\_absolute\_url(self):  
 return reverse('category', kwargs={'cat\_slug': self.slug})  
  
  
class TagPost(models.Model):  
 tag = models.CharField(max\_length=100, db\_index=True)  
 slug = models.SlugField(max\_length=255, unique=True, db\_index=True)  
 author = models.ForeignKey(get\_user\_model(), on\_delete=models.SET\_NULL,null=True, related\_name='tagposts')  
  
 def \_\_str\_\_(self):  
 return self.tag  
  
 def get\_absolute\_url(self):  
 return reverse('tag', kwargs={'tag\_slug': self.slug})  
  
  
class UploadFiles(models.Model):  
 file = models.FileField(upload\_to='uploads\_model')

В файле описаны четыре модели:

Модель **Records** представляет собой запись (пост) с различными аттрибутами, такими как заголовок, рейтинг, фото и т. д. Эта модель также включает функционал для управления статусом публикации, создания и редактирования записей.

Поля:

**rating**: Числовое поле для оценки записи, ограниченное значениями от 0 до 10 с использованием валидаторов MinValueValidator и MaxValueValidator.

**title**: Заголовок записи, строковое поле с максимальной длиной 255 символов.

**slug**: Уникальный URL-совместимый идентификатор записи (slug), создающийся на основе заголовка.

**photo**: Поле для загрузки фотографии, где изображения сохраняются в директории photos/%Y/%m/%d/. Поле может быть пустым (blank=True) и может быть null.

**content**: Основное текстовое содержимое записи.

**time\_create**: Время создания записи, автоматически заполняемое при создании.

**time\_update**: Время последнего изменения записи, автоматически обновляется при каждом изменении.

**is\_published**: Флаг, определяющий, опубликована ли запись или нет. Используются поля выбора, где 1 означает "Активно" (опубликовано), а 0 — "В корзине" (неопубликовано).

**cat**: Связь с категорией через внешний ключ. При удалении категории, это поле устанавливается в null.

**tags**: Связь с тегами через ManyToManyField. Это позволяет записи быть связанными с несколькими тегами.

**author**: Внешний ключ, ссылающийся на модель пользователя, который является автором записи. Если пользователь удаляется, это поле становится null.

**Менеджеры**:

**objects**: Базовый менеджер для всех объектов.

**published**: Специальный менеджер, который возвращает только опубликованные записи. Используется фильтрация по полю **is\_published**.

Методы:

**\_\_str\_\_():** Строковое представление объекта, возвращает заголовок записи.

**get\_absolute\_url():** Возвращает URL для просмотра записи на основе её slug.

Метаданные:

**verbose\_name**: Читаемое имя модели в единственном числе — "пост".

**verbose\_name\_plural**: Читаемое имя модели во множественном числе — "посты".

**ordering**: Устанавливает порядок сортировки по умолчанию (по убыванию рейтинга и времени создания).

**indexes**: Создает индекс по полю time\_create для ускорения запросов.

Модель **Category** представляет собой категории, к которым могут относиться записи. Каждая категория имеет имя и уникальный slug, а также связь с пользователем.

Поля:

**name**: Название категории.

**slug**: Уникальный идентификатор категории в URL.

**author**: Внешний ключ, ссылающийся на модель пользователя, который создал категорию.

Метаданные:

**verbose\_name**: Читаемое имя модели в единственном числе — "категория".

**verbose\_name\_plural**: Читаемое имя модели во множественном числе — "категории".

Модель **TagPost** используется для хранения тегов, которые могут быть привязаны к записям. Она также имеет связь с пользователями.

Поля:

**tag**: Название тега.

**slug**: Уникальный идентификатор тега в URL.

**author**: Внешний ключ, ссылающийся на модель пользователя, который создал тег.

**Модель UploadFiles** предназначена для загрузки файлов, связанных с записью. Файлы сохраняются в директории uploads\_model.

Поля:

**file**: Поле для загрузки файлов. Указывает путь для загрузки файлов в директорию uploads\_model.

- Метод \_\_str\_\_ отдаёт строку с номером, датой окончания подачи заявок и предметом закупки.

- Метод end\_submission() проверяет, прошел ли срок подачи заявок с использованием срока в 90 дней (может быть изменено).

- Также определены метаданные для отображения в админке.

Эти модели (классы) позволяют удобно управлять и отслеживать созданные пользователем записи в БД. Записи могут быть добавлены, обновлены и архивированы, что помогает поддерживать порядок в данных.

Файл директории ./records/forms.py

Назначение и роль файла forms.py в Django:

1. Обработка данных от пользователя: Файл forms.py используется для обработки данных, полученных от пользователей. В нем описываются формы для ввода данных, такие как добавление записей, категорий, тегов или файлов. Все формы проверяются и валидируются перед тем, как данные будут сохранены в базе данных.

2. Валидация данных: Форма в Django позволяет не только собирать данные от пользователя, но и проверять их правильность с помощью встроенных и кастомных методов валидации. Это важно для обеспечения целостности данных и предотвращения ошибок.

3. Упрощение работы с моделями: Формы в Django автоматически связываются с моделями, что позволяет легко извлекать и сохранять данные в базе данных. Через forms.ModelForm Django упрощает взаимодействие с моделями, предоставляя готовую структуру для создания, редактирования и удаления объектов.

4. Интерфейс пользователя: Используя поля типа TextInput, Textarea, ChoiceField, Django формирует удобный интерфейс для ввода данных. Атрибуты виджетов, такие как widget.attrs, позволяют кастомизировать внешний вид и сделать формы более удобными для пользователя.

from django.core.exceptions import ValidationError  
from .models import Category, Records, TagPost  
from django import forms  
  
  
class AddPostForm(forms.ModelForm):  
 def \_\_init\_\_(self, \*args, \*\*kwargs):  
 user = kwargs.pop('user', None) # Получаем текущего пользователя  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
  
 if user:  
 # Фильтруем категории и теги по текущему пользователю  
 self.fields['cat'].queryset = Category.objects.filter(author=user)  
 self.fields['tags'].queryset = TagPost.objects.filter(author=user)  
  
 # Изменяем метку для поля tags  
 self.fields['tags'].label = 'Выберите темы'  
  
 # Применяем классы для полей  
 self.fields['rating'].widget.attrs.update({  
 'class': 'form-select rating', # Добавляем класс для rating  
 })  
  
 self.fields['cat'].widget.attrs.update({  
 'class': 'form-select cat', # Добавляем класс для cat  
 })  
  
 self.fields['is\_published'].widget.attrs.update({  
 'class': 'form-select is-published', # Добавляем класс для is\_published  
 })  
  
 self.fields['tags'].widget.attrs.update({  
 'class': 'form-select tags', # Добавляем класс для tags  
 })  
  
 cat = forms.ModelChoiceField(queryset=Category.objects.all(), empty\_label="Категория не выбрана", label="Категории")  
 tags = forms.ModelMultipleChoiceField(queryset=TagPost.objects.all(), required=False)  
  
 rating = forms.ChoiceField(  
 choices=[(str(i), str(i)) for i in range(1, 11)],  
 label="Рейтинг"  
 )  
  
 class Meta:  
 model = Records  
 fields = ['title', 'content', 'photo', 'rating', 'is\_published', 'cat', 'tags']  
 widgets = {  
 'title': forms.TextInput(attrs={'class': 'form-input'}),  
 'content': forms.Textarea(attrs={'cols': 50, 'rows': 5}),  
 }  
  
 def clean\_rating(self):  
 rating = int(self.cleaned\_data.get('rating'))  
 if rating < 1 or rating > 10:  
 raise ValidationError("Рейтинг должен быть от 1 до 10")  
 return rating  
  
 def clean\_title(self):  
 title = self.cleaned\_data['title']  
 if len(title) > 50:  
 raise ValidationError("Длина превышает 50 символов")  
 return title  
  
 def clean(self):  
 cleaned\_data = super().clean()  
 title = cleaned\_data.get('title')  
 user = self.instance.author # assuming you have an 'author' field in your model, and it's set correctly  
 if Records.objects.filter(title=title, author=user).exists():  
 raise ValidationError("У вас уже есть запись с таким названием")  
 return cleaned\_data  
  
  
class BaseCategoryTagForm(forms.ModelForm):  
 def \_\_init\_\_(self, \*args, \*\*kwargs):  
 self.user = kwargs.pop('user', None)  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
  
 def clean\_field(self, field\_name, model, user\_field, error\_message):  
 value = self.cleaned\_data[field\_name]  
  
 # Проверка длины  
 if len(value) > 50:  
 raise ValidationError("Длина превышает 50 символов")  
  
 # Проверка уникальности для текущего пользователя  
 if model.objects.filter(\*\*{field\_name: value, user\_field: self.user}).exists():  
 raise ValidationError(error\_message)  
  
 return value  
  
  
from .models import Category, TagPost  
  
  
class AddCategoryForm(BaseCategoryTagForm):  
 class Meta:  
 model = Category  
 fields = ['name']  
 widgets = {  
 'name': forms.TextInput(attrs={'class': 'form-input'}),  
 }  
  
 def clean\_name(self):  
 return self.clean\_field('name', Category, 'author', "Категория с таким названием уже существует.")  
  
  
class AddTagForm(BaseCategoryTagForm):  
 class Meta:  
 model = TagPost  
 fields = ['tag']  
 widgets = {  
 'tag': forms.TextInput(attrs={'class': 'form-input'}),  
 }  
  
 def clean\_tag(self):  
 return self.clean\_field('tag', TagPost, 'author', "Тег с таким названием уже существует.")  
  
  
class UploadFileForm(forms.Form):  
 file = forms.ImageField(label="Файл")

*Классы форм и их поля:*

*1. AddPostForm* - Это форма для добавления новой записи в систему, которая использует модель Records. Она позволяет пользователю заполнять несколько полей и отправить данные в базу. Рассмотрим поля этой формы:

cat = forms.ModelChoiceField:

**ModelChoiceField** — это поле, которое отображает выпадающий список с возможными вариантами для выбора одного объекта из модели. В этом поле пользователь выбирает категорию для своей записи. Поле связано с моделью Category, которая содержит список всех категорий.

Параметры:

**queryset=Category.objects.all():** Это означает, что все объекты модели Category будут отображены в списке для выбора.

**empty\_label="Категория не выбрана":** Если пользователь не выбрал категорию, в списке будет отображаться этот текст.

**label="Категории":** Это текст, который будет отображаться рядом с полем, чтобы пользователь знал, что ему нужно выбрать категорию.

tags = forms.ModelMultipleChoiceField:

**ModelMultipleChoiceField** — это поле для выбора нескольких объектов из набора, то есть пользователю дается возможность выбрать несколько тегов для записи. Поле позволяет пользователю выбрать несколько тегов для записи. Поле связано с моделью **TagPost**.

Параметры:

**queryset=TagPost.objects.all():** Это указывает на то, что все объекты модели TagPost (теги) будут отображаться для выбора.

**required=False**: Это означает, что поле необязательно для заполнения. Пользователь может оставить его пустым, если не хочет выбирать теги.

rating = forms.ChoiceField:

**ChoiceField** — поле выбора с заранее определенными вариантами. Поле позволяет пользователю выбрать рейтинг для записи. Доступные значения — числа от 1 до 10.

Параметры:

choices=[(str(i), str(i)) for i in range(1, 11)]: Это список вариантов, где каждый элемент представляет собой кортеж с двумя одинаковыми значениями: строковым представлением числа. Например, вариант будет (1, '1'), (2, '2') и так далее до 10.

**label="Рейтинг"** - текст, который будет отображаться рядом с полем в шаблоне.

title = forms.TextInput:

**TextInput** — это поле для ввода текста. Поле для ввода названия записи. Это обязательное поле для заполнения, поскольку каждая запись должна иметь уникальное название.

Параметры:

attrs={'class': 'form-input'}: Это атрибуты HTML для стилизации поля (например, добавляется класс CSS для улучшения внешнего вида).

content = forms.Textarea:

**Textarea** — это поле для ввода многострочного текста. Поле для ввода содержимого записи. Это большое текстовое поле, которое позволяет ввести описание или текст записи.

Параметры:

attrs={'cols': 50, 'rows': 5}: Эти параметры устанавливают размеры поля — 50 символов по ширине и 5 строк по высоте.

is\_published = forms.ChoiceField:

**ChoiceField** - поле позволяет пользователю выбрать, будет ли запись опубликована. Это может быть полезно для записи черновиков или предварительных версий.

Параметры:

В параметрах не указано явно, что это поле представляет собой булев выбор ("Да" или "Нет"), но на практике, вероятно, это так. Здесь могут быть значения, такие как True/False или другие, которые контролируют, опубликована ли запись.

В этой форме, **AddPostForm,** определены методы для валидации данных:

**clean\_rating** - метод проверяет введенный рейтинг. Он уверяет, что значение находится в диапазоне от 1 до 10. Если рейтинг выходит за пределы этого диапазона, выбрасывается ошибка с сообщением "Рейтинг должен быть от 1 до 10".

**clean\_titl** - метод проверяет длину заголовка записи. Если длина заголовка превышает 50 символов, выбрасывается ошибка с сообщением "Длина превышает 50 символов".

**clean** - метод выполняет общую проверку формы. Он проверяет, есть ли уже запись с таким названием у текущего пользователя. Если такая запись существует, выбрасывается ошибка с сообщением "У вас уже есть запись с таким названием".

*2. BaseCategoryTagForm* - это абстрактная форма, которая используется для создания категорий и тегов. Она содержит общую логику валидации и проверки уникальности.

**clean\_field** - метод, который используется для проверки значений полей на длину и уникальность. Метод проверяет:

Длину значения поля. Если длина превышает 50 символов, выбрасывается ошибка.

Уникальность значения для текущего пользователя. Он проверяет, существует ли уже объект с таким значением, и если да — выбрасывает ошибку с заданным сообщением.

*3. AddCategoryForm* - форма для добавления новой категории, для ввода её названия. Название должно быть уникальным для каждого пользователя.

name = forms.TextInput:

**clean\_name** - проверяет, что длина названия не превышает 50 символов и что категория с таким названием уже не существует для данного пользователя.

*4. AddTagForm* **-** для ввода названия тега. Тег также должен быть уникальным для каждого пользователя.

tag = forms.TextInput:

**clean\_tag** - проверяет, что длина тега не превышает 50 символов и что тег с таким названием уже не существует для данного пользователя.

*5. UploadFileForm* - для загрузки изображения. Пользователь может выбрать и загрузить файл изображения.

file = forms.ImageField:

**label="Файл":** Текст, который будет отображаться рядом с полем загрузки, чтобы пользователи знали, что им нужно загрузить изображение.

Файл ./records/admin.py

Файл «admin.py» в Django используется для настройки административного интерфейса приложения. В этом файле можно зарегистрировать модели, настроить их отображение и добавить дополнительные функции, такие как фильтры и поиск.

from django.contrib import admin, messages  
from .models import Records, Category, TagPost  
  
  
@admin.register(Records)  
class RecordsAdmin(admin.ModelAdmin):  
 list\_display = ('id', 'title', 'time\_create', 'cat', 'is\_published', 'breaf\_info')  
 list\_display\_links = ('id', 'time\_create')  
 ordering = ('id', 'time\_create', 'is\_published')  
 list\_editable = ['is\_published', 'title']  
 list\_per\_page = 10  
 actions = ['set\_published', 'set\_delete', ]  
 search\_fields = ['title', 'cat\_\_name', ]  
 list\_filter = ['cat\_\_name', 'is\_published']  
  
 @admin.display(description='Длина поста')  
 def breaf\_info(self, record: Records):  
 return f"{len(record.content)} символов"  
  
 @admin.action(description='Убрать из корзины')  
 def set\_published(self, request, queryset):  
 queryset.update(is\_published=Records.Status.PUBLISHED)  
 self.message\_user(request, f"{queryset.count()} постов возвращено из корзины.")  
  
 @admin.action(description='Переместить в корзину')  
 def set\_delete(self, request, queryset):  
 queryset.update(is\_published=Records.Status.DRAFT)  
 self.message\_user(request, f"{queryset.count()} постов помещено в корзину.", messages.WARNING)  
  
  
@admin.register(Category)  
class CategoryAdmin(admin.ModelAdmin):  
 list\_display = ('id', 'name', 'slug')  
 list\_display\_links = ('id', 'name')  
  
  
  
@admin.register(TagPost)  
class TagPostAdmin(admin.ModelAdmin):  
 list\_display = ('id', 'tag', 'slug')

Описание файла admin.py

Файл admin.py в проекте Django используется для настройки административного интерфейса, который позволяет администратору управлять моделями проекта через веб-интерфейс. В этом файле настраиваются следующие аспекты:

Настройка модели **Records**:

**list\_display**: Определяет поля, которые будут отображаться в списке объектов модели в админке. Например, id, title, time\_create, cat (категория), is\_published (статус публикации).

**list\_display\_links**: Указывает, какие поля будут кликабельными для перехода к редактированию объекта. В данном случае это поля id и time\_create.

**ordering**: Устанавливает порядок сортировки объектов. Здесь записи сортируются по id, затем по времени создания и статусу публикации.

**list\_editable**: Позволяет редактировать определенные поля прямо в списке объектов. В этом случае это поля is\_published и title.

**list\_per\_page**: Ограничивает количество объектов, отображаемых на одной странице в админке (в данном случае 10).

**actions**: Определяет доступные действия, которые можно выполнить с выбранными объектами. Например, действия set\_published и set\_delete для изменения статуса публикации

**search\_fields**: Указывает, по каким полям будет производиться поиск. В данном случае поиск возможен по полям title и cat\_\_name.

**list\_filter**: Добавляет фильтры для быстрого поиска объектов. В данном случае это фильтры по категории и статусу публикации.

Настройка модели **Category**:

Класс **CategoryAdmin** отвечает за отображение и управление категориями в административном интерфейсе.

**list\_display**: Включает поля id, name (название категории) и slug (который является уникальным идентификатором категории).

**list\_display\_links**: Указывает, что поля id и name будут кликабельными для перехода к редактированию.

Настройка модели **TagPost**:

Класс **TagPostAdmin** управляет отображением и редактированием тегов, которые могут быть связаны с записями.

**list\_display**: Показывает поля id, tag (сам тег) и slug (унитарный идентификатор тега).

**list\_display\_links**: Поля id и tag будут кликабельными для редактирования тега.

Файл admin.py в Django проекте предоставляет гибкую настройку административного интерфейса, позволяя администратору удобно управлять записями, категориями и тегами. С помощью различных настроек, таких как list\_display, actions и search\_fields, администратор может легко работать с большими объемами данных в админке.

Таким образом, код настраивает админку Django, позволяя удобно управлять данными процедур и архива с функциями импорта и экспорта.

Файл ./records/views.py

Файл views.py в Django отвечает за обработку запросов и возвращение ответов.

Вот основные функции views.py:

\* Обработка запросов. Определяет, что делать с входящими HTTP-запросами (GET, POST и т.д.).

\* Логика приложения. Содержит бизнес-логику, связанную с обработкой данных.

\* Взаимодействие с моделями. Получает данные из базы данных через модели и передает их в шаблоны.

\* Отправка ответов. Формирует и возвращает HTTP-ответы (как страницы HTML, JSON и т.д.).

Таким образом, views.py является связующим звеном между моделями и интерфейсом пользователя

from django.contrib.auth.mixins import LoginRequiredMixin  
from django.core.paginator import Paginator  
from django.http import HttpResponse, HttpResponseNotFound, Http404, HttpResponseRedirect, HttpResponsePermanentRedirect  
from django.shortcuts import render, redirect, get\_object\_or\_404  
from django.urls import reverse, reverse\_lazy  
from django.views import View  
from django.views.generic import TemplateView, ListView, DetailView, FormView, CreateView, UpdateView, DeleteView  
from .forms import AddPostForm, UploadFileForm, AddCategoryForm, AddTagForm  
from .models import Records, Category, TagPost, UploadFiles  
from .utils import DataMixin  
from django.utils.text import slugify  
from unidecode import unidecode  
from django.contrib.auth.decorators import login\_required  
  
  
def generate\_slug(text):  
 *"""Функция генерации слага на основе переданного ей текста."""* return slugify(unidecode(text))  
  
  
class RecordsHome(LoginRequiredMixin, DataMixin, ListView):  
 template\_name = 'records/index.html'  
 context\_object\_name = 'posts'  
 cat\_selected = 0  
 extra\_context = {'title': 'Главная страница'}  
  
 def get\_queryset(self):  
 return Records.objects.filter(author=self.request.user).select\_related('cat')  
  
  
# Краткое описание о возможностях сайта.  
class AboutSite(TemplateView):  
 template\_name = 'records/about.html'  
 extra\_context = {'title': 'О сайте'}  
  
  
class ShowPost(DataMixin, DetailView):  
 template\_name = 'records/post.html'  
 slug\_url\_kwarg = 'post\_slug'  
 context\_object\_name = 'post'  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 return self.get\_mixin\_context(context, title=context['post'].title)  
  
 def get\_object(self, queryset=None):  
 return get\_object\_or\_404(Records.objects, slug=self.kwargs[self.slug\_url\_kwarg])  
  
  
class AddPost(LoginRequiredMixin, DataMixin, CreateView):  
 form\_class = AddPostForm  
 template\_name = 'records/addpost.html'  
 extra\_context = {'title': 'Добавить статью'}  
  
 def form\_valid(self, form):  
 w = form.save(commit=False)  
 w.author = self.request.user  
 w.slug = generate\_slug(w.title)  
 return super().form\_valid(form)  
  
  
class UpdatePost(DataMixin, UpdateView):  
 model = Records  
 fields = ['title', 'content', 'rating', 'photo', 'is\_published', 'cat', 'tags', ]  
 template\_name = 'records/addpost.html'  
 success\_url = reverse\_lazy('home')  
 extra\_context = {'title': 'Редактировать запись'}  
  
  
class Contacts(TemplateView):  
 template\_name = 'records/contacts.html'  
 extra\_context = {  
 'title': 'Контакты',  
 'phone\_number': 'Впишите сюда номер телефона',  
 'email': 'Впишите сюда e-mail адрес',  
 'address': 'Впишите сюда адрес',  
 }  
  
  
class RecordsCategory(DataMixin, ListView):  
 template\_name = 'records/index.html'  
 context\_object\_name = 'posts'  
 allow\_empty = False  
  
 def get\_queryset(self):  
 return Records.objects.filter(cat\_\_slug=self.kwargs['cat\_slug']).select\_related("cat")  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 cat = context['posts'][0].cat  
 return self.get\_mixin\_context(context,  
 title=cat.name,  
 cat\_selected=cat.pk,  
 )  
  
  
class TagPostList(DataMixin, ListView):  
 template\_name = 'records/index.html'  
 context\_object\_name = 'posts'  
 allow\_empty = True  
  
 def get\_queryset(self):  
 # Фильтруем записи по категории и текущему пользователю  
 queryset = Records.objects.filter(  
 cat\_\_slug=self.kwargs['cat\_slug'],  
 author=self.request.user # Фильтрация по текущему пользователю  
 ).select\_related("cat")  
 return queryset  
  
 def get\_context\_data(self, \*, object\_list=None, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 tag = TagPost.objects.get(slug=self.kwargs['tag\_slug'])  
 return self.get\_mixin\_context(context, title=tag.tag)  
  
 def get\_queryset(self):  
 return Records.published.filter(tags\_\_slug=self.kwargs['tag\_slug']).select\_related('cat')  
  
  
# Для удаления записи из таблицы  
class DeletePost(View):  
 def post(self, request, slug):  
 post = get\_object\_or\_404(Records, slug=slug)  
  
 # Проверка на подтверждение  
 if request.POST.get('confirm\_delete') == 'yes':  
 post.delete()  
 return redirect('home') # Перенаправление на список постов (или на нужную вам страницу)  
  
 return redirect('post', post\_slug=post.slug)  
  
  
# Измменение поля "is\_published"  
class ToggleFlagView(View):  
 def post(self, request, slug):  
 record = get\_object\_or\_404(Records, slug=slug)  
 status = not record.is\_published  
 record.is\_published = status  
 record.save()  
 title = ["Переместили в корзину", "Убрали из корзины"][status]  
 return render(request, 'records/post.html', {'post': record, 'title': title})  
  
  
class AddCategory(LoginRequiredMixin, CreateView):  
 form\_class = AddCategoryForm  
 template\_name = 'records/addcategory.html'  
 extra\_context = {'title': 'Добавить категорию'}  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 # Передаем категорию и форму с текущим пользователем  
 context['categories'] = Category.objects.filter(author=self.request.user)  
 context['form'] = self.get\_form()  
 return context  
  
 def get\_form\_kwargs(self):  
 *"""Передаем пользователя в форму."""* kwargs = super().get\_form\_kwargs()  
 kwargs['user'] = self.request.user # передаем пользователя в форму  
 return kwargs  
  
 def form\_valid(self, form):  
 category = form.save(commit=False)  
 category.author = self.request.user  
 base\_slug = generate\_slug(category.name)  
 slug = slugify(base\_slug)  
 counter = 1  
 while Category.objects.filter(slug=slug).exists():  
 slug = f"{base\_slug}{counter}"  
 counter += 1  
 category.slug = slug  
  
 category.save()  
 return redirect(reverse('addcategory'))  
  
  
class CategoryUpdateView(LoginRequiredMixin, UpdateView):  
 model = Category  
 form\_class = AddCategoryForm  
 template\_name = 'records/addcategory.html'  
  
 def get\_object(self):  
 return get\_object\_or\_404(Category, slug=self.kwargs['slug'], author=self.request.user)  
  
 def get\_success\_url(self):  
 return reverse('addcategory')  
  
  
class AddTag(LoginRequiredMixin, CreateView):  
 form\_class = AddTagForm  
 template\_name = 'records/addtag.html'  
 extra\_context = {'title': 'Добавить тему'}  
  
 def get\_context\_data(self, \*\*kwargs):  
 context = super().get\_context\_data(\*\*kwargs)  
 # Передаем категорию и форму с текущим пользователем  
 context['data'] = TagPost.objects.filter(author=self.request.user)  
 context['form'] = self.get\_form()  
 return context  
  
 def get\_form\_kwargs(self):  
 *"""Передаем пользователя в форму."""* kwargs = super().get\_form\_kwargs()  
 kwargs['user'] = self.request.user # передаем пользователя в форму  
 return kwargs  
  
 def form\_valid(self, form):  
 tag = form.save(commit=False)  
 tag.author = self.request.user  
 base\_slug = generate\_slug(tag.tag)  
 slug = slugify(base\_slug)  
 counter = 1  
 while TagPost.objects.filter(slug=slug).exists():  
 slug = f"{base\_slug}{counter}"  
 counter += 1  
 tag.slug = slug  
  
 tag.save()  
 return redirect(reverse('addtag'))  
  
  
class TagUpdateView(LoginRequiredMixin, UpdateView):  
 model = TagPost  
 form\_class = AddTagForm  
 template\_name = 'records/addtag.html'  
  
 def get\_object(self):  
 return get\_object\_or\_404(TagPost, slug=self.kwargs['slug'], author=self.request.user)  
  
 def get\_success\_url(self):  
 return reverse('addtag')  
  
  
@login\_required  
def delete\_object(request, slug, model):  
 if model == 'category':  
 obj = get\_object\_or\_404(Category, slug=slug, author=request.user)  
 obj.delete()  
 return redirect('addcategory')  
 elif model == 'tag':  
 obj = get\_object\_or\_404(TagPost, slug=slug, author=request.user)  
 obj.delete()  
 return redirect('addtag')  
 else:  
 # Возвращаем ошибку или делаем что-то, если передана неизвестная модель  
 return redirect('home')

**Рассмотрим логику кода подробнее:**

Импортируемые модули

**from django.contrib.auth.mixins import LoginRequiredMixin**: Этот миксин используется для ограничения доступа к определённым представлениям только для авторизованных пользователей. Если пользователь не авторизован, его перенаправляют на страницу входа.

**from django.core.paginator import Paginator**: Используется для пагинации списка записей. Это позволяет разбивать большой список на страницы, чтобы пользователи не загружали слишком много данных за раз.

**from django.shortcuts import render, redirect, get\_object\_or\_404**: Эти функции используются для рендеринга страниц (функция render), перенаправления на другие страницы (функция redirect), а также для безопасного извлечения объектов из базы данных (функция get\_object\_or\_404).

**from django.urls import reverse, reverse\_lazy**: Модули для работы с URL-адресами. Функция reverse используется для получения URL по имени представления, а reverse\_lazy — для отложенной генерации URL (например, в случае с переадресацией после успешного сохранения формы).

**from django.views import View**: Это базовый класс для создания представлений (views), которые могут обрабатывать HTTP-запросы. Используется для создания собственных представлений, не ограничиваясь стандартными представлениями (например, ListView или DetailView).

**from django.views.generic import TemplateView, ListView, DetailView, CreateView, UpdateView, DeleteView**: Это классические представления Django, которые обеспечивают стандартное поведение для рендеринга шаблонов (TemplateView), отображения списка объектов (ListView), детальной информации о записи (DetailView), создания объектов (CreateView), обновления объектов (UpdateView) и их удаления (DeleteView).

**Модели и формы:**

from .forms import AddPostForm, UploadFileForm, AddCategoryForm, AddTagForm: Импортируются формы для добавления записей (AddPostForm), загрузки файлов (UploadFileForm), создания категорий (AddCategoryForm) и тегов (AddTagForm).

**from .models import Records, Category, TagPost, UploadFiles**: Эти модели определяют структуру базы данных для записей, категорий, тегов и загружаемых файлов. Каждая модель соответствует таблице в базе данных.

**from django.utils.text import slugify**: Модуль для генерации "slug" (уникальных идентификаторов) для записей и категорий на основе текста.

**from unidecode import unidecode**: Модуль для нормализации текста: удаляет акценты и специальные символы, что упрощает создание slug'ов.

**Пользовательские модули и миксины:**

**from .utils import DataMixin**: Это кастомный миксин, который добавляет общие данные в контекст каждого представления (например, добавляет титулы для страниц или другие общие параметры).

Для работы с запросами и ответами:

**from django.http import HttpResponse, HttpResponseNotFound, Http404, HttpResponseRedirect, HttpResponsePermanentRedirect**: Эти классы отвечают за создание различных типов HTTP-ответов, таких как стандартные ответы, ошибки (например, 404) или редиректы.

Краткая логика начала работы с проектом

В данном проекте реализован сайт для хранения записей, ссылок, фото с возможностью задания рейтинга и других меток, а также системой авторизации, чтобы пользователи могли работать с индивидуальными данными. Основные компоненты проекта описаны в файле views.py, который отвечает за обработку запросов и взаимодействие с моделями проекта.

**Главная страница и доступ к записям:**

Класс **RecordsHome** отображает список постов текущего пользователя. Он наследует от **LoginRequiredMixin**, что означает, что только авторизованные пользователи могут получить доступ к странице. Здесь также используется пагинация для удобства работы с большим количеством записей.

Добавление, редактирование и удаление записей:

Классы **AddPost**, **UpdatePost** и **DeletePost** позволяют пользователям добавлять, редактировать и удалять свои записи. При создании новой записи автоматически генерируется уникальный slug (идентификатор), а записи можно редактировать, изменяя их содержимое, категорию, рейтинг и статус публикации.

Просмотр записи:

Класс **ShowPost** используется для отображения подробной информации о записи. Каждая запись доступна по своему уникальному URL, основанному на slug.

Категории и теги:

В проекте также реализована система категорий и тегов. Классы **AddCategory**, **CategoryUpdateView**, **AddTag** и **TagUpdateView** позволяют пользователям добавлять и редактировать категории и теги, которые могут быть присвоены записям. Пользователи могут фильтровать записи по этим категориям и тегам.

Контакты и информация о сайте:

Класс **Contacts** отвечает за страницу с контактной информацией, а класс **AboutSite** предоставляет краткую информацию о сайтe.

**Изменение статуса публикации записи:**

Класс **ToggleFlagView** позволяет изменять статус публикации записи, переводя её в статус "активно" или "в корзине".

Обработка действий с записями:

Функции, такие как delete\_object, используются для удаления записей, категорий и тегов через запросы, отправленные от пользователя.

Основные моменты, которые стоит учитывать при запуске проекта:

Перед использованием сайта необходимо зарегистрировать пользователя и пройти процесс авторизации.

Убедитесь, что установлены все зависимости, указанные в requirements.txt, чтобы проект корректно работал.

Взаимодействие с записями, категориями и тегами осуществляется через административный интерфейс или страницы сайта.

Все изменения, касающиеся контента (например, создание, редактирование и удаление записей), доступны только авторизованным пользователям, что обеспечивает безопасность данных.

Файл records/urls.py

Этот код определяет маршруты URL для Django-приложения.

from django.urls import path  
from . import views  
  
  
urlpatterns = [  
 path('', views.RecordsHome.as\_view(), name='home'),  
 path('about/', views.AboutSite.as\_view(), name='about'),  
 path('contacts/', views.Contacts.as\_view(), name='contacts'),  
 path('addpost/', views.AddPost.as\_view(), name='addpost'),  
 path('post/<slug:post\_slug>/', views.ShowPost.as\_view(), name='post'),  
 path('tag/<slug:tag\_slug>/', views.TagPostList.as\_view(), name='tag'),  
 path('update/<slug:slug>/', views.UpdatePost.as\_view(), name='update\_post'),  
 path('delete/<slug:slug>/',views.DeletePost.as\_view() , name='delete\_post'),  
 path('toggle\_flag/<slug:slug>/', views.ToggleFlagView.as\_view(), name='toggle\_flag'),  
 path('category/<slug:cat\_slug>/', views.RecordsCategory.as\_view(), name='category'),  
 path('addcategory/', views.AddCategory.as\_view(), name='addcategory'),  
 path('editcategory/<slug:slug>/', views.CategoryUpdateView.as\_view(), name='editcategory'),  
 path('addtag/', views.AddTag.as\_view(), name='addtag'),  
 path('edittag/<slug:slug>', views.TagUpdateView.as\_view(), name='edittag'),  
 path('deletecategory/<slug:slug>/', views.delete\_object, {'model': 'category'}, name='deletecategory'),  
 path('deletetag/<slug:slug>/', views.delete\_object, {'model': 'tag'}, name='deletetag'),  
]

Сначала мы импортируем функцию path для создания маршрутов и views для подключения представлений, которые будут обрабатывать запросы. Указываем имя приложения, чтобы разделять пространства имен для URL-адресов. Это полезно при использовании reverse или include.

Определяем маршруты:

- "": Корневой маршрут, который связывается с представлением home и имеет имя home.

- "about/": Маршрут для страницы "О сайте", связывается с представлением about.

- "contacts/": маршрут для отображения контактных данных.

- "post//": Маршрут для отображения поста по slug.

- "/": Маршрут для отображение главной страницы.

Таким образом, каждый маршрут связан с соответствующим представлением, которое будет обрабатывать HTTP-запросы и возвращать необходмые страницы или данные.

**Шаблон стартовой страницы** (./templates/records/index)

{% extends 'base.html' %}  
  
{% block content %}  
<ul class="list-articles">  
 {% for p in posts %}  
 <li><div class="article-panel">  
 <p class="first">Категория: {{p.cat.name}}, рэйтинг {{ p.rating }}</p>  
 <p class="last">Последние изменения: {{p.time\_update|date:"d-m-Y H:i:s"}}</p>  
 </div>  
{% if p.photo %}  
 <p><img class="img-article-left thumb" src="{{p.photo.url}}"></p>  
{% endif %}  
  
 <h2>{{p.title}}</h2>  
 {% autoescape off %}  
 {{p.content|linebreaks|truncatewords:40}}  
 {% endautoescape %}  
 <div class="clear"></div>  
 <p class="link-read-post">  
 <a href="{{ p.get\_absolute\_url }}" class="btn">Подробнее</a>  
 </p>  
 </li>  
 {% endfor %}  
</ul>  
{% endblock %}  
  
  
{% block navigation %}  
{% if page\_obj.has\_other\_pages %}  
<nav class="list-pages">  
 <ul>  
 {% if page\_obj.has\_previous %}  
 <li class="page-num">  
 <a href="?page={{ page\_obj.previous\_page\_number }}">&lt;</a>  
 </li>  
 {% endif %}  
  
 {% for p in paginator.page\_range %}  
 {% if page\_obj.number == p %}  
 <li class="page-num page-num-selected">{{ p }}</li>  
 {% elif p >= page\_obj.number|add:-2 and p <= page\_obj.number|add:2 %}  
 <li class="page-num">  
 <a href="?page={{ p }}">{{ p }}</a>  
 </li>  
 {% endif %}  
 {% endfor %}  
  
 {% if page\_obj.has\_next %}  
 <li class="page-num">  
 <a href="?page={{ page\_obj.next\_page\_number }}">&gt;</a>  
 </li>  
 {% endif %}  
  
 </ul>  
</nav>  
{% endif %}  
{% endblock %}

Расширение базового шаблона - {% extends "base.html" %} указывает, что данный шаблон наследует содержимое из base.html о котором было упомянуто ранее.

{% block title %}{{title}}{% endblock %} – определяем блок title, который будет заменен на значение переменной title, переданное из представления в базовый(base.html), родительский, шаблон.

Далее в блоке с контентом страницы выводятся данные полученные из БД(записи пользователя).

По итогу код создает страницу с показом информации о дате последнего изменения возможного выбора постов по связанным категориям или темам. так же выводится краткое содержание и рейтинг записи.

**Шаблон ./templates/records/about.html**

Страница предназначена для информации о сайте, особенностями работы с ним и т.д..

{% extends 'base.html' %}  
  
{% block content %}  
<p>Данный сайт предназначен для хранения статей/ссылок/фотографий  
 с целью группировки по тематическим описаниям, а так же возможностью задания ценности информации.</p>  
{% endblock %}

Шаблон так же наследуется от базового с изменением блока title и блока content.

**Шаблон ./templates/addpost.html**

Шаблон позволяет ять записи в БД и сохранять их там согласно критериям указанным в форме, а так же загружать изображения.

{% extends 'base.html' %}  
  
{% block content %}  
<h1>{{ title\_page }}</h1>  
<form action="" method="post" enctype="multipart/form-data">  
 {% csrf\_token %}  
 <div class="form-error">{{ form.non\_field\_errors }}</div>  
 {% for f in form %}  
 <p><label class="form-label" for="{{ f.id\_for\_label }}">{{ f.label }}</label>{{ f }}</p>  
 <div class="form-error">{{ f.errors }}</div>  
 {% endfor %}  
 <p>  
 <button type="submit">Сохранить</button>  
 </p>  
</form>  
{% endblock %}

Шаблон так же наследуется от базового с изменением блока title и блока content.

На странице расположена кнопка «Сохранить», которая вызывает функцию save() , для обновления списка новых опубликованных процедур (форма, отправляющая POST-запрос для обновления записи в БД, защищенная токеном CSRF).

Далее выводится главная страница с добавленной записью.

**Шаблон ./templates/addcategory.html**

{% extends 'base.html' %}  
  
{% block content %}  
<h1>{{ title\_page }}</h1>  
<form action="" method="post" enctype="multipart/form-data">  
 {% csrf\_token %}  
 <div class="form-error">{{ form.non\_field\_errors }}</div>  
 {% for f in form %}  
 <p><label class="form-label" for="{{ f.id\_for\_label }}">{{ f.label }}</label>{{ f }}</p>  
 <div class="form-error">{{ f.errors }}</div>  
 {% endfor %}  
 <p>  
 <button type="submit">Сохранить</button>  
 </p>  
</form>  
  
<h3>Список категорий:</h3>  
<ul>  
 {% for category in categories %}  
 <li class="category-item">  
 <span class="category-name">{{ category.name }}</span>  
 <a href="{% url 'editcategory' category.slug %}" class="btn btn-warning">Редактировать</a>  
 <form action="{% url 'deletecategory' category.slug %}" method="post" style="display:inline;">  
 {% csrf\_token %}  
 <button type="submit" class="btn btn-danger">Удалить</button>  
 </form>  
 </li>  
 {% empty %}  
 <li>Вы еще не добавили ни одной категории.</li>  
 {% endfor %}  
</ul>  
  
{% endblock %}

Переходя на данную страницу будут выведены все категории с возможностью редактировать/удалять уже имеющиеся записи или вносить новые категории, группирующие записи.

Директория с миграциями (migrations)

Директория migrations в приложении Django используется для управления изменениями в базе данных, связанными с моделями. Она содержит файлы миграций, которые отслеживают изменения в структуре базы данных (например, добавление или изменение полей в моделях).

Данная директория создается автоматически при миграции моделей (создании таблиц в базе данных) с параметрами, заданными при разработке приложения.

**Приложение users**

Приложение users отвечает за регистрацию новых пользователей, отображение информации о существующих пользователях, а также их настройку.

Файл ./users/apps.py

from django.apps import AppConfig  
  
  
class UsersConfig(AppConfig):  
 default\_auto\_field = 'django.db.models.BigAutoField'  
 name = 'users'  
 verbose\_name = "Пользователи"

Описание полей:

- default\_auto\_field: устанавливает тип автоматически создаваемого идентификатора для моделей. В данном случае используется BigAutoField, что позволяет использовать более широкий диапазон значений для ID.

- name: указывает имя приложения в Django. Здесь оно обозначено как "users".

- verbose\_name: определяет понятное и человеко-читаемое имя приложения, которое будет отображаться в админ-панели и других местах. В данном случае это "Пользователи".

**Файл /.users/forms.py**

from django import forms  
from django.contrib.auth import get\_user\_model  
from django.contrib.auth.forms import AuthenticationForm, UserCreationForm, PasswordChangeForm  
  
  
class LoginUserForm(AuthenticationForm):  
 username = forms.CharField(label="Логин",  
 widget=forms.TextInput(attrs={'class': 'form-input'}))  
 password = forms.CharField(label="Пароль",  
 widget=forms.PasswordInput(attrs={'class': 'form-input'}))  
  
 class Meta:  
 model = get\_user\_model()  
 fields = ['username', 'password']  
  
  
class RegisterUserForm(UserCreationForm):  
 username = forms.CharField(label="Логин", widget=forms.TextInput(attrs={'class': 'form-input'}))  
 date\_of\_birth = forms.DateField(label="Дата рождения", widget=forms.DateInput(attrs={'class': 'form-input'}))  
 password1 = forms.CharField(label="Пароль", widget=forms.PasswordInput(attrs={'class': 'form-input'}))  
 password2 = forms.CharField(label="Повтор пароля", widget=forms.PasswordInput(attrs={'class': 'form-input'}))  
  
 class Meta:  
 model = get\_user\_model()  
 fields = ['username', 'email', 'first\_name', 'last\_name', 'date\_of\_birth','password1', 'password2']  
 labels = {  
 'email': 'E-mail',  
 'first\_name': "Имя",  
 'last\_name': "Фамилия",  
 }  
 widgets = {  
 'email': forms.TextInput(attrs={'class': 'form-input'}),  
 'first\_name': forms.TextInput(attrs={'class': 'form-input'}),  
 'last\_name': forms.TextInput(attrs={'class': 'form-input'}),  
 }  
  
 def clean\_email(self):  
 email = self.cleaned\_data['email']  
 if get\_user\_model().objects.filter(email=email).exists():  
 raise forms.ValidationError("Такой E-mail уже существует!")  
 return email  
  
  
class ProfileUserForm(forms.ModelForm):  
 username = forms.CharField(disabled=True, label='Логин', widget=forms.TextInput(attrs={'class': 'form-input'}))  
 email = forms.CharField(disabled=True, label='E-mail', widget=forms.TextInput(attrs={'class': 'form-input'}))  
  
 class Meta:  
 model = get\_user\_model()  
 fields = ['username', 'email', 'first\_name', 'last\_name']  
 labels = {  
 'first\_name': 'Имя',  
 'last\_name': 'Фамилия',  
 }  
 widgets = {  
 'first\_name': forms.TextInput(attrs={'class': 'form-input'}),  
 'last\_name': forms.TextInput(attrs={'class': 'form-input'}),  
 }  
  
  
class UserPasswordChangeForm(PasswordChangeForm):  
 old\_password = forms.CharField(label="Старый пароль", widget=forms.PasswordInput(attrs={'class': 'form-input'}))  
 new\_password1 = forms.CharField(label="Новый пароль", widget=forms.PasswordInput(attrs={'class': 'form-input'}))  
 new\_password2 = forms.CharField(label="Подтверждение пароля", widget=forms.PasswordInput(attrs={'class': 'form-input'}))

**Описание форм для регистрации и управления пользователями в Django.**

В данном модуле реализованы несколько форм, которые отвечают за взаимодействие с пользователями в приложении. Эти формы касаются регистрации пользователей, их аутентификации, редактирования профиля и изменения пароля. Каждая из форм построена с использованием возможностей Django, что делает их удобными для интеграции в веб-приложение с базой данных и с возможностью легко настраивать внешнее оформление через стилизацию. Важно отметить, что все формы используют стандартные механизмы валидации данных, что гарантирует правильность введенной информации.

*1. Форма для входа в систему (LoginUserForm)*

Форма LoginUserForm используется для аутентификации пользователей на сайте. Она позволяет зарегистрированным пользователям вводить свои учетные данные (логин и пароль), чтобы получить доступ к защищенным разделам сайта. Данная форма наследуется от встроенной формы AuthenticationForm, которая является стандартным решением для входа в систему в Django.

Поля формы:

Логин (поле username): Пользователь должен ввести свой логин, который был указан при регистрации. Это обязательное поле, которое является строковым и отображается как текстовое поле (TextInput). Для улучшения пользовательского интерфейса и удобства ввода, это поле стилизуется с помощью CSS-класса form-input.

Пароль (поле password): Пользователь должен ввести свой пароль. Для обеспечения конфиденциальности введенные символы скрыты, и это поле отображается как PasswordInput. Также для стилизации используется класс form-input.

Метаданные:

Модель: get\_user\_model(), которая возвращает текущую модель пользователя, используемую в проекте. Это может быть стандартная модель пользователя Django или кастомная модель.

Поля: ['username', 'password'], которые являются обязательными для авторизации.

Данная форма выполняет стандартную проверку данных, а именно проверку правильности введенного логина и пароля, и в случае успешной аутентификации предоставляет пользователю доступ к защищенным разделам сайта.

*2. Форма регистрации нового пользователя (RegisterUserForm)*

Форма RegisterUserForm предназначена для регистрации нового пользователя в системе. Эта форма наследует от стандартной формы UserCreationForm, которая включает в себя базовую логику для создания учетной записи пользователя в Django. Однако форма была дополнена рядом дополнительных полей и валидаций.

Поля формы:

Логин (поле username): Поле для ввода уникального имени пользователя. Логин является обязательным для регистрации и должен быть уникальным в системе.

Дата рождения (поле date\_of\_birth): Это поле позволяет пользователю указать свою дату рождения при регистрации. Виджет для этого поля — DateInput, который отображает календарь для удобного выбора даты.

E-mail (поле email): Поле для ввода электронной почты пользователя. Это поле обязательно и должно содержать уникальный адрес электронной почты. Оно также проверяется на уникальность в базе данных, чтобы избежать регистрации нескольких аккаунтов с одинаковыми адресами.

Имя (поле first\_name): Поле для ввода имени пользователя.

Фамилия (поле last\_name): Поле для ввода фамилии пользователя.

Пароль (поле password1): Первое поле для ввода пароля.

Подтверждение пароля (поле password2): Поле для повторного ввода пароля, чтобы подтвердить правильность ввода.

Форма включает в себя валидацию:

Валидация пароля: Проверяется, совпадают ли введенные пароли в полях password1 и password2. Если они не совпадают, будет выведена ошибка.

Валидация e-mail: В методе clean\_email выполняется проверка на уникальность введенного e-mail. Если такой адрес уже существует в базе данных, будет выведена ошибка с сообщением о том, что данный e-mail уже зарегистрирован.

Метаданные:

Модель: get\_user\_model(), которая используется для работы с пользователями в проекте.

Поля: ['username', 'email', 'first\_name', 'last\_name', 'date\_of\_birth', 'password1', 'password2'].

Метки и виджеты: Для удобства пользователя заданы метки для полей и применены виджеты с CSS-классом form-input для всех полей, что помогает стилизовать форму и сделать её более привлекательной и понятной.

*3. Форма редактирования профиля пользователя (ProfileUserForm)*

Форма ProfileUserForm предоставляет пользователю возможность редактировать личные данные, такие как имя и фамилию, но логин и e-mail отображаются только для чтения. Это необходимо для того, чтобы пользователь не мог изменить данные, которые являются уникальными и должны оставаться неизменными, например, логин или e-mail.

Поля формы:

Логин (поле username): Это поле отображается в форме, но оно является только для чтения (атрибут disabled=True), что значит, что пользователь не может изменить свой логин через эту форму.

E-mail (поле email): Это поле также отображается в форме, но оно также заблокировано для редактирования.

Имя (поле first\_name) и Фамилия (поле last\_name): Эти поля доступны для редактирования пользователем. Виджеты для этих полей — TextInput, с атрибутом class='form-input', который позволяет красиво стилизовать текстовые поля.

Метаданные:

Модель: get\_user\_model(), которая возвращает текущую модель пользователя, используемую в проекте.

Поля: ['username', 'email', 'first\_name', 'last\_name'].

Эта форма полезна, когда пользователю необходимо только обновить определенные данные, не касаясь чувствительных, таких как логин и e-mail.

*4. Форма изменения пароля (UserPasswordChangeForm)*

Форма UserPasswordChangeForm используется для смены пароля пользователя. Она наследует от стандартной формы PasswordChangeForm, предоставляемой Django. Эта форма позволяет пользователю безопасно обновить свой пароль.

Поля формы:

Старый пароль (поле old\_password): Поле для ввода текущего пароля пользователя, который нужно ввести для подтверждения своей личности.

Новый пароль (поле new\_password1): Поле для ввода нового пароля.

Подтверждение нового пароля (поле new\_password2): Поле для повторного ввода нового пароля, чтобы убедиться в правильности ввода.

Все поля для ввода пароля используют виджет PasswordInput, который скрывает введенные символы. Это важный момент, так как он обеспечивает конфиденциальность введенных данных.

Метаданные:

Модель: get\_user\_model().

Поля: ['old\_password', 'new\_password1', 'new\_password2'].

Форма гарантирует, что новый пароль введен правильно и подтвержден пользователем. Она также защищает данные, вводимые в поля пароля, через использование виджетов с маскировкой символов.

**Файл ./users/views.py**

Отвечает за аутентификацию пользователей в приложении и включает в себя представления для входа, регистрации и выхода из системы.

from django.contrib.auth import get\_user\_model, logout  
from django.contrib.auth.mixins import LoginRequiredMixin  
from django.contrib.auth.views import LoginView, PasswordChangeView  
from django.shortcuts import redirect  
from django.urls import reverse, reverse\_lazy  
from django.views.generic import CreateView, UpdateView  
  
from .forms import LoginUserForm, RegisterUserForm, ProfileUserForm, UserPasswordChangeForm  
  
  
class LoginUser(LoginView):  
 form\_class = LoginUserForm  
 template\_name = 'users/login.html'  
 extra\_context = {'title': 'Авторизация'}  
  
  
class RegisterUser(CreateView):  
 form\_class = RegisterUserForm  
 template\_name = 'users/register.html'  
 extra\_context = {'title': "Регистрация"}  
 success\_url = reverse\_lazy('users:login')  
  
  
class ProfileUser(LoginRequiredMixin, UpdateView):  
 model = get\_user\_model()  
 form\_class = ProfileUserForm  
 template\_name = 'users/profile.html'  
 extra\_context = {'title': "Профиль пользователя"}  
  
 def get\_success\_url(self):  
 return reverse\_lazy('users:profile')  
  
 def get\_object(self, queryset=None):  
 return self.request.user  
  
  
class UserPasswordChange(PasswordChangeView):  
 form\_class = UserPasswordChangeForm  
 success\_url = reverse\_lazy("users:password\_change\_done")  
 template\_name = "users/password\_change\_form.html"  
  
  
def logout\_user(request):  
 logout(request)  
 return redirect('users:login')

*1. Представление для входа (LoginUser)*

Представление LoginUser наследуется от встроенного представления LoginView в Django и используется для аутентификации пользователей. Оно позволяет пользователю войти в систему, предоставив свои учетные данные (логин и пароль). При успешной аутентификации пользователь будет перенаправлен на главную страницу сайта или в защищенные разделы.

Параметры:

form\_class: указывает на форму, которая будет использоваться для входа пользователя. В данном случае это форма LoginUserForm, которая описана в forms.py.

template\_name: указывает путь к шаблону, который будет использован для отображения страницы входа. В данном случае это шаблон users/login.html.

extra\_context: передает дополнительную информацию в контекст шаблона. В данном случае добавляется переменная title, которая будет отображать заголовок страницы как "Авторизация".

Логика: При получении POST-запроса с данными пользователя, форма выполняет проверку на правильность введенных данных. Если данные верны, пользователь получает доступ к системе.

*2. Представление для регистрации пользователя (RegisterUser)*

Представление RegisterUser используется для создания новой учетной записи пользователя на сайте. Оно наследуется от класса CreateView, который предоставляет функциональность для создания нового объекта в базе данных.

Параметры:

form\_class: указывает на форму для регистрации пользователя. В данном случае используется форма RegisterUserForm, которая определена в forms.py.

template\_name: указывает путь к шаблону, который будет использоваться для отображения страницы регистрации. Шаблон users/register.html используется для этого представления.

extra\_context: передает дополнительную информацию в контекст шаблона. В данном случае добавляется переменная title, которая будет отображать заголовок страницы как "Регистрация".

success\_url: это URL, на который будет перенаправлен пользователь после успешной регистрации. В данном случае используется reverse\_lazy('users:login'), что перенаправляет пользователя на страницу логина.

Логика: Форма регистрации создает нового пользователя в базе данных с указанными данными, такими как логин, e-mail, имя, фамилия и дата рождения. После успешной регистрации пользователь перенаправляется на страницу входа в систему.

*3. Представление для редактирования профиля (ProfileUser)*

Представление ProfileUser позволяет пользователю редактировать свой профиль. Оно наследуется от UpdateView, который предоставляет функциональность для обновления существующего объекта в базе данных. В данном случае пользователю предоставляется возможность обновить свои личные данные, такие как имя и фамилия.

Параметры:

model: указывает модель, с которой работает представление. В данном случае это модель пользователя, возвращаемая функцией get\_user\_model().

form\_class: указывает на форму, которая будет использоваться для редактирования профиля. Здесь используется форма ProfileUserForm, которая описана в forms.py.

template\_name: указывает путь к шаблону, который будет использоваться для отображения страницы редактирования профиля. Это шаблон users/profile.html.

extra\_context: добавляет дополнительную переменную title в контекст шаблона, которая будет отображаться как "Профиль пользователя".

Методы:

get\_success\_url: возвращает URL, на который будет перенаправлен пользователь после успешного обновления данных. В данном случае это reverse\_lazy('users:profile'), что возвращает пользователя обратно на страницу редактирования профиля.

get\_object: этот метод позволяет получить объект, который будет редактироваться. Здесь он возвращает текущего пользователя, что позволяет редактировать только свой профиль.

Логика: Пользователь может обновить свои данные, такие как имя и фамилия, но не может изменить логин или e-mail. После успешного обновления данных, пользователь будет перенаправлен обратно на страницу профиля.

*4. Представление для изменения пароля (UserPasswordChange)*

Представление UserPasswordChange предназначено для изменения пароля пользователя. Оно наследуется от встроенного представления PasswordChangeView, которое предоставляет функциональность для смены пароля.

Параметры:

form\_class: указывает на форму для смены пароля. В данном случае это форма UserPasswordChangeForm, которая определена в forms.py.

success\_url: URL, на который пользователь будет перенаправлен после успешного изменения пароля. В данном случае это reverse\_lazy("users:password\_change\_done"), что приведет пользователя на страницу, информирующую о завершении изменения пароля.

template\_name: указывает путь к шаблону для отображения страницы изменения пароля. Это шаблон users/password\_change\_form.html.

Логика: Форма позволяет пользователю ввести старый пароль, новый пароль и подтвердить новый пароль. После успешного обновления пароля, пользователь будет перенаправлен на страницу, которая сообщает ему о успешной смене пароля.

*5. Представление для выхода из системы (logout\_user)*

Представление logout\_user отвечает за выход пользователя из системы. Оно вызывает встроенную функцию logout из Django, которая завершает сессию текущего пользователя.

Логика:

После выполнения функции logout(request), текущий пользователь будет выведен из системы, и его сессия будет завершена.

После выхода пользователь будет перенаправлен на страницу входа в систему с помощью redirect('users:login'), но он будет иметь просматривать возможность по адресам "./contacts/", "./about/", a так же "./home/" но без отображение данных из БД.

**Файл ./users/urls.py**

Определяет маршруты (URLs) для приложения Django, связанного с управлением пользователями.

from django.contrib.auth.views import LogoutView, PasswordChangeView, PasswordChangeDoneView  
from django.urls import path  
from . import views  
  
app\_name = "users"  
  
urlpatterns = [  
 path('login/', views.LoginUser.as\_view(), name='login'),  
 path('logout/', views.logout\_user, name='logout'),  
  
 path('password-change/', views.UserPasswordChange.as\_view(), name="password\_change"),  
 path('password-change/done/', PasswordChangeDoneView.as\_view(template\_name="users/password\_change\_done.html"), name="password\_change\_done"),  
  
 path('register/', views.RegisterUser.as\_view(), name='register'),  
 path('profile/', views.ProfileUser.as\_view(), name='profile'),  
]

Импорт необходимых модулей: функция path для создания маршрутов, представления (views) из текущего модуля (.).

Определили пространство имен для приложения, что позволяет ссылаться на маршруты через user\_app:login и т.д.

- path("login/", views.LoginUser.as\_view(), name="login"): маршрут для входа пользователя. Используется класс-представление LoginUser для обработки запроса.

- path("registration/", views.UserRegistrationView.as\_view(), name="registration"): маршрут для регистрации нового пользователя. Обрабатывается классом UserRegistrationView.

- path('password-change/',views.UserPasswordChange.as\_view(), name="password\_change") Этот маршрут указывает на страницу для изменения пароля. Когда пользователь посещает путь /password-change/, вызывается представление UserPasswordChange, которое предоставляет форму для смены пароля.

- path('password-change/done/', PasswordChangeDoneView.as\_view(template\_name="users/password\_change\_done.html"), name="password\_change\_done")

Этот маршрут ведет на страницу, которая отображается после успешного изменения пароля.

При переходе по пути /password-change/done/, вызывается встроенное представление PasswordChangeDoneView, которое отображает страницу с подтверждением, что пароль был успешно изменен.

Используется шаблон users/password\_change\_done.html, который информирует пользователя о завершении операции.

Название маршрута (name="password\_change\_done") важно для ссылок, которые могут направлять пользователя на страницу завершения изменения пароля.

- path('register/', views.RegisterUser.as\_view(), name='register') маршрут ведет на страницу регистрации нового пользователя.

- path('profile/', views.ProfileUser.as\_view(), name='profile') маршрут ведет на страницу профиля пользователя.

Пользователь может редактировать свои данные (например, имя и фамилию) через представление ProfileUser, которое использует форму для обновления данных пользователя.

- path("logout/", views.logout\_user, name="logout"): маршрут для выхода пользователя. В этом случае используется функция logout из представлений.

**Шаблон ./templates/users/login.htm**

{% extends 'base.html' %}  
  
{% block content %}  
<h1>Авторизация</h1>  
  
<form method="post">  
 {% csrf\_token %}  
 <input type="hidden" name="next" value="{{ next }}" />  
 <div class="form-error">{{ form.non\_field\_errors }}</div>  
 {% for f in form %}  
 <p><label class="form-label" for="{{ f.id\_for\_label }}">{{ f.label }}</label>{{ f }}</p>  
 <div class="form-error">{{ f.errors }}</div>  
 {% endfor %}  
 <button type="submit" class="btn">Войти</button>  
</form>  
{% endblock %}

Этот код представляет собой шаблон для страницы входа (авторизации) пользователя, поскольку реализована проверка как через логи так и через почту авторизация становится более удобна.

Данный шаблон наследует содержимое из base.html. Это позволяет использовать общие элементы оформления, определенные в базовом шаблоне. Создаем блок для заголовка страницы. Значение {{ title }} будет заменено на конкретное название страницы, переданное из представления.

В блоке контента создаётся форма, которая отправляется на URL, соответствующий представлению для входа (users:login. Здесь также добавлен токен CSRF для защиты от атак. Если есть ошибки ввода, они будут отображаться в виде красного уведомления.

Внизу под формой добавляем ссылку для перехода на страницу регистрации нового аккаунта.

Как итог этот шаблон создает страницу для входа пользователя с обработкой ошибок, использованием форм и с возможностью перехода на страницу регистрации. Он оформлен с помощью Bootstrap и наследует общий стиль из базового шаблона.

**Шаблон ./templates/users/registration.html**

{% extends 'base.html' %}  
  
{% block content %}  
<h1>Регистрация</h1>  
  
<form method="post">  
 {% csrf\_token %}  
 <input type="hidden" name="next" value="{{ next }}" />  
 <div class="form-error">{{ form.non\_field\_errors }}</div>  
 {% for f in form %}  
 <p><label class="form-label" for="{{ f.id\_for\_label }}">{{ f.label }}</label>{{ f }}</p>  
 <div class="form-error">{{ f.errors }}</div>  
 {% endfor %}  
 <p><button type="submit" class="btn">Регистрация</button></p>  
</form>  
  
{% endblock %}

Этот шаблон отвечает за отображение формы регистрации пользователя на веб-странице.

Он также наследуется от базового шаблона base.html с применением стилей и передачей заголовка title.

Форма отправляется по URL, который соответствует представлению registration в приложении users. Метод отправки — POST. Включает защиту от подделки межсайтовых запросов.

Далее отображаются поля формы. Каждое поле формы отображается с соответствующей меткой. Если есть ошибки валидации, они будут выведены в блоке предупреждения.

Данный шаблон предоставляет пользователю удобный интерфейс для регистрации, обрабатывает ввод данных и проверяет наличие ошибок, чтобы обеспечить корректное заполнение формы для сохранения данных в БД.

**Шаблон ./templates/users/profile.html**

{% extends 'base.html' %}  
  
{% block content %}  
<h1>Профиль</h1>  
<div id="action-buttons">  
 <form method="post">  
 {% csrf\_token %}  
 <div class="form-error">{{ form.non\_field\_errors }}</div>  
 {% for f in form %}  
 <p><label class="form-label" for="{{ f.id\_for\_label }}">{{ f.label }}: </label>{{ f }}</p>  
 <div class="form-error">{{ f.errors }}</div>  
 {% endfor %}  
  
 <div class="form-buttons">  
 <button type="submit" class="btn">Сохранить</button>  
 <a href="/users/password-change/" class="btn">Изменить пароль</a>  
 </div>  
 </form>  
</div>  
  
{% endblock %}

Здесь пользователь сможет посмотреть свои данные и при необходимости перейти на вкладку смены пароля

**Шаблон ./templates/users/password\_change\_form.html**

{% extends 'base.html' %}  
  
{% block content %}  
<h1>Изменение пароля</h1>  
<div>  
 <form method="post">  
 {% csrf\_token %}  
 <div class="form-error">{{ form.non\_field\_errors }}</div>  
 {% for f in form %}  
 <p><label class="form-label" for="{{ f.id\_for\_label }}">{{f.label}}: </label>{{ f }}</p>  
 <div class="form-error">{{ f.errors }}</div>  
 {% endfor %}  
 <button type="submit" class="btn">Изменить пароль</button>  
 </p>  
 </form>  
 <a href="/users/profile/" class="btn">Назад</a>  
</div>  
{% endblock %}

Доступна смена пароля или возврат к профилю. Так же он может вернуться на главную или посмотреть группы записей, поскольку все маршруты "Все категории", "Добавить запись" являются гиперссылками и доступны в наследуемом базовом шаблоне.

**Заключение**

В заключении дипломной работы, посвященной разработке сайта на базе Python и Django для эффективного сохранения и организации пользовательских данных, можно выделить несколько ключевых аспектов.

Разработанный сайт предоставляет пользователю уникальную возможность хранить статьи, фотографии и ссылки, при этом обеспечивая гибкую структуру для их группировки по темам и категориям. Система позволяет пользователю устанавливать рейтинг своим записям, что способствует улучшению взаимодействия с контентом и повышению уровня персонализации. Важно отметить, что доступ к этим данным ограничен только автором, что гарантирует конфиденциальность и безопасность информации.

Использование Python и Django обеспечило платформе высокую гибкость и масштабируемость, что особенно актуально для динамично развивающихся областей. Автоматизация ряда процессов значительно снизила временные затраты на выполнение рутинных операций, а также повысила точность обработки данных. Это позволило улучшить качество и объем хранимой информации, а также минимизировать время, необходимое для выполнения различных операций.

Внедрение предложенной системы дает пользователям возможность более эффективно управлять своими записями, легко находить и структурировать информацию по своим предпочтениям. Результаты работы подтверждают значимость интеграции современных технологий для оптимизации процессов управления данными и открывают широкие перспективы для дальнейших разработок в этой области. В процессе испольльзования сайта, разработанного на базе Django, позволит значительно повысить эффективность работы с данными.

Список используемой литературы

* Лекции с программы обучения «Python-разработчик. Специалист»
* “Python. Разработка на основе тестирования.” Гарри Персиваль
* “Python, Django и PyCharm для начинающих.” Постолит А. В.
* Стандартная библиотека Python. Документация.
* Документация Django 4
* Документация django-import-export
* Документация bootstrap
* Интернет статьи по теме

Приложения

Весь код и файлы: <https://github.com/Apizh/Site-for-Records>