**ФГАОУ ВО «МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

Отчет по курсовому проекту по дисциплине

«Инженерное проектирование»

Тема: «Проектирование веб-приложения на серверной стороне»

Выполнил

студент 1 курса

группы 231-322

Супроткин Р.В.

Проверила

Старший преподаватель

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Даньшина М.В.

**МОСКВА 2023**

**Содержание**

1. Введение
2. Предметная область
3. База данных: таблицы и записи
   1. Модели **Event** и **Project**
   2. Модель **CustomUser**
4. Django туториал
5. CRUD: пример и реализация
   1. Create
   2. Read
   3. Update
   4. Delete
6. Django панель администратора

**1. Введение**

Курсовая работа представляет собой веб-приложение на React.js и Django, которое представляет собой портфолио студентов (проекты, результаты участия в мероприятиях). В нём реализованы возможности просмотра профилей всех зарегистрированных пользователей, которые классифицируются студентами ВУЗов или сотрудниками высших учебных заведений или компаний. Как студенты, так и сотрудники могут организовывать мероприятия или создавать проекты, на которые могут записываться пользователи. Реализована возможность создавать проекты, изменять их данные, включая число участников и организаторов, а также удалять записи об этих активностях (то есть удалять их).

Структура проекта полена на 2 части:

* Frontend – папка, в которой хранится код «внешней» части веб-приложения, то есть того, что видит пользователь. Она написана на React.js. Выбор в пользу именно этой библиотеки был сделан исходя из высокой эффективности React в приложениях, где часто нужно «отрисовывать» те или иные компоненты (например, в чат-приложениях или стриминговых сайтах. Twitch написан на React, а также музыкальный плеер vk). В этом приложении часто выполняется рендеринг компонентов, отвечающих за «карточки» пользователей или активностей.
* Backend – папка, в которой хранится код «внутренней» части веб-приложения, то есть того, что происходит на сервере. Фреймворк Django был выбран в силу отсутствия права выбора

Курсовая работы выполнена студентом 1 курса ФГАОУ ВО «МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ» Супроткином Русланом Васильевичем, принадлежащим к студенческой группе номер 231-322. Процесс выполнения работы был зафиксирован при помощи системы контроля версий Git, ссылка на репозиторий: <https://github.com/RuslanitoSS/kursach>

1. **Предметная область**

Предполагается, что приложением будут пользоваться студенты ВУЗов и сотрудники высших учебных заведений или компаний. Как студенты, так и сотрудники характеризуются:

* ID
* Именем
* Фамилией
* Отчеством (при наличии)
* Полом (М/Ж)
* Адресом эл. Почты
* Названием места работы / ВУЗом обучения

Исходя из этого было принято решения создать одну модель пользователя, в которой будут храниться данных как об студентах, так и об сотрудниках. Для реализации этой задумки нужно добавить поле, отвечающее за тип пользователя (студент или сотрудник)

Как студенты, так и сотрудники могут организовывать мероприятия или создавать проекты. Проекты характеризуются следующими качествами:

* ID
* Названием
* Кратким описанием
* Полным описанием
* Участниками (поле M2M к сущности, отвечающей за пользователя)
* Организаторами (поле M2M к сущности, отвечающей за пользователя)

Мероприятия характеризуются следующими качествами:

* ID
* Названием
* Кратким описанием
* Полным описанием
* Адресом проведения
* Датой начала
* Датой завершения
* Участниками (поле M2M к сущности, отвечающей за пользователя)
* Организаторами (поле M2M к сущности, отвечающей за пользователя)

Организаторы имеют право изменять данные проектов и мероприятий, включая число участников и организаторов, а также удалять записи об этих активностях.

1. **База данных: таблицы и записи**

В современном мире программирования и разработки веб-приложений использование баз данных — это неотъемлемая часть любого приложения, поскольку данные должны где-то храниться, обрабатываться и структурироваться. В контексте веб-разработки с использованием фреймворка Django, таблицы и записи в базе данных играют ключевую роль в организации данных и их структурировании для дальнейшего взаимодействия с пользователями и логикой приложения.

**Таблицы в базе данных**

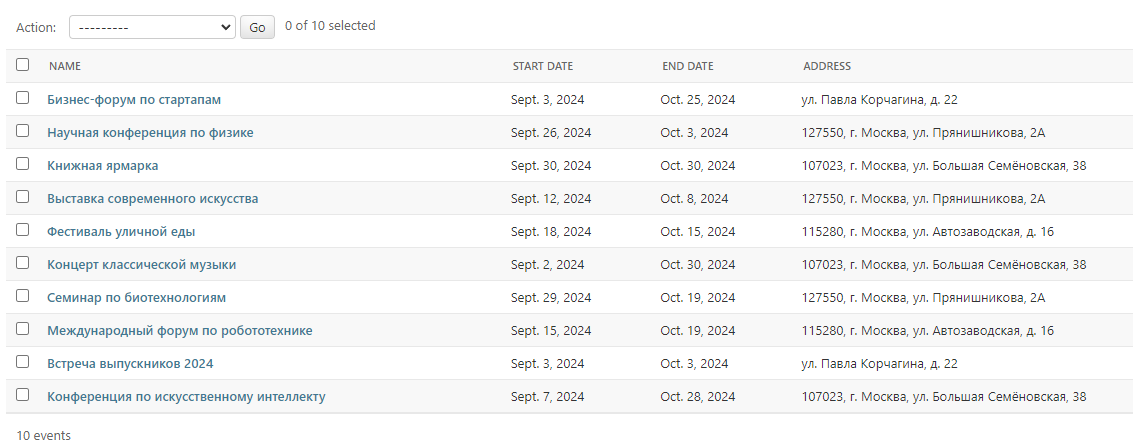
В Django, каждая таблица является моделью — это класс, который описывает структуру и характеристики данных, которые будут храниться в таблице. Каждая модель в Django автоматически преобразуется в таблицу базы данных с аналогичным названием, где поля модели соответствуют столбцам в таблице.

**Модель Event**

Таблица для модели **Event** будет содержать следующие столбцы:

1. **id** — Автоматически создаваемый уникальный идентификатор для каждой записи. В базах данных он часто используется как первичный ключ (primary key), позволяющий уникально идентифицировать каждую запись.
2. **name** — Поле для хранения названия события, представляющее собой текстовое значение, максимальная длина которого ограничена 255 символами. Это важный столбец, поскольку он содержит основную информацию, позволяющую пользователям различать события.
3. **description** — Текстовое поле, которое используется для хранения полного описания события. Оно не ограничено по длине и может содержать значительные объемы текста.
4. **short\_description** — Краткое описание события, аналогичное полю name, но, возможно, содержащее более сжатую и полезную информацию для быстрого обзора.
5. **address** — Этот столбец хранит адрес, где происходит событие, и может быть пустым (опциональным) для записей, где это не указано.
6. **start\_date** и **end\_date** — Даты начала и окончания события. **start\_date** является обязательным полем, а **end\_date** может быть пустым.
7. **participants** и **organizers** — Множество участников и организаторов события, которые представлены через связь "многие ко многим" (ManyToManyField). В базах данных это означает, что одна запись в таблице событий может быть связана с множеством записей в таблице пользователей, и наоборот. Для этой связи создаётся вспомогательная таблица.

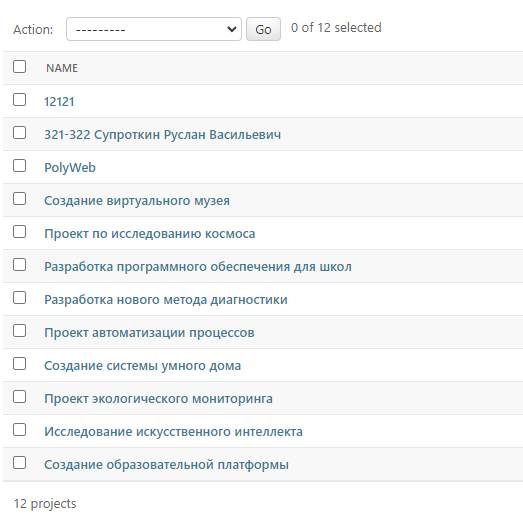
Таким образом, каждая строка в таблице **Event** представляет собой отдельное событие с уникальными характеристиками, и каждая запись может включать множество участников и организаторов.



*10 нетестовых записей в таблице* ***Event***

**Модель Project**

Модель **Project** практически идентична модели Event с точки зрения структуры, но она представляет собой другую сущность в системе. Поля name, description, short\_description, participants и organizers аналогичны полям модели Event, но они относятся к проектам, а не событиям. Структура таблицы для **Project** аналогична структуре таблицы для **Event**, но с акцентом на проекты.



*10 нетестовых (и 2 тестовые) записей в таблице* ***Project***

**Записи в базе данных**

Запись в базе данных — это одна строка в таблице, содержащая данные о конкретной сущности. Когда создаётся новая запись в таблице **Event**, система автоматически генерирует уникальный идентификатор в поле id, и каждый из столбцов, таких как name, description и start\_date, заполняется соответствующими значениями. После этого данная запись сохраняется в базе данных и становится доступной для дальнейшего использования — например, отображения в интерфейсе, модификации или удаления.

При работе с базой данных важна концепция ссылок и связей между различными таблицами. В моделях **Events** и **Projects** есть тип связи М2М с моделью пользователя (**CustomUser**).

1. **Связь "Многие ко многим" (ManyToManyField)** — Поля participants и organizers в моделях Event и Project представляют связь "многие ко многим" с таблицей пользователей (**CustomUser**). Это означает, что одно событие или проект может иметь множество участников, и один пользователь может участвовать во множестве событий или проектов. Для управления этими связями создаются промежуточные таблицы, в которых хранятся соответствия между событиями (или проектами) и пользователями.

**Модель пользователя (CustomUser)**

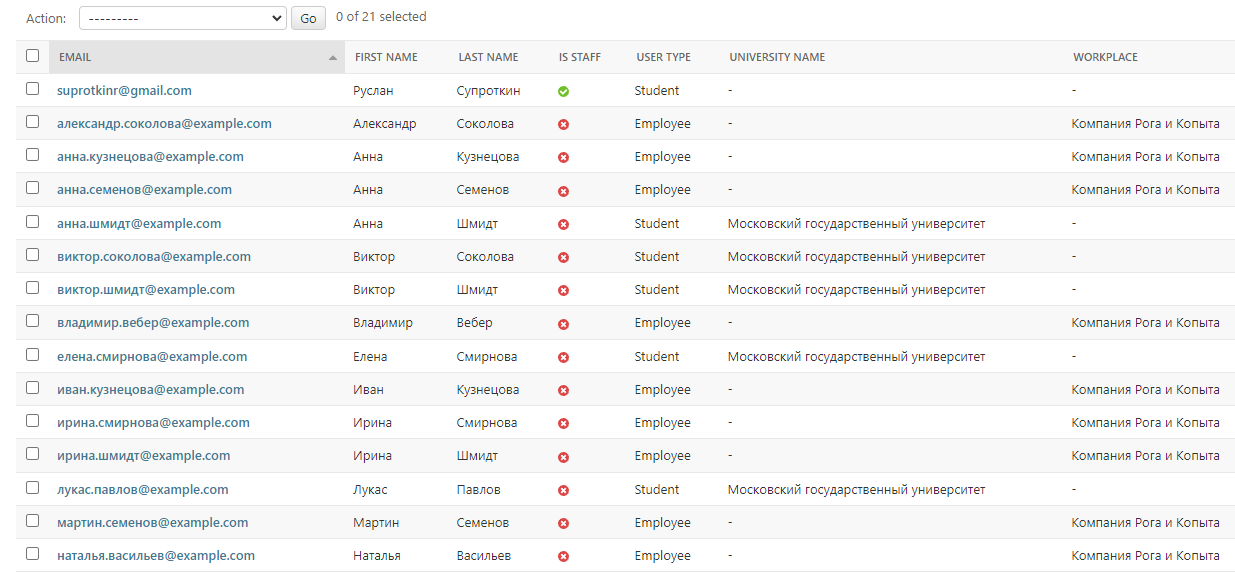
Модель пользователя — это одна из важнейших сущностей в любом веб-приложении, поскольку она отвечает за хранение данных о пользователях. В данном веб приложении используется кастомная модель пользователя, которая расширяет стандартные возможности, предоставляемые Django, добавляя дополнительные поля и особенности. Именно такой подход к хранению данных о пользователях был выбран в силу анализа предметной области. Так как полей стандартной модели **User** оказалось недостаточно для полной реализации темы, было принято решение создать свою модель пользователя. Класс **CustomUser**, который наследуется от AbstractBaseUser и PermissionsMixin, создаёт основу для новой таблицы в базе данных, где будет храниться информация о пользователях.

**Таблица CustomUser**

Каждый класс модели в Django преобразуется в таблицу базы данных. Таблица **CustomUser** будет содержать следующие столбцы, которые отражают поля модели:

1. email — Это поле, представляющее электронную почту пользователя. Оно уникально, что означает, что в базе данных не может быть двух записей с одинаковыми значениями в этом столбце. Это одно из ключевых полей, которое используется в качестве идентификатора пользователя вместо стандартного поля username, используемого по умолчанию в Django.
2. first\_name, middle\_name, last\_name — Поля для хранения имени, отчества и фамилии пользователя. Они могут быть пустыми, что даёт гибкость в ситуациях, когда не вся информация предоставлена при регистрации.
3. is\_active — Булево поле, определяющее, активен ли пользователь. Активные пользователи могут входить в систему и взаимодействовать с приложением. Неактивные пользователи могут быть заблокированы или удалены, но их данные остаются в базе.
4. is\_staff — Булевое поле, указывающее, является ли пользователь сотрудником или администратором. Пользователи с этим флагом имеют доступ к административной части сайта (Django admin).
5. date\_joined — Поле, содержащее информацию о дате и времени, когда пользователь присоединился к приложению. Это поле автоматически заполняется текущей датой и временем при создании новой записи.
6. gender — Поле, представляющее пол пользователя, оно использует предопределённые варианты выбора (Male, Female, Other), что реализовано с помощью choices. Это даёт возможность легко организовать данные и избежать некорректных значений.
7. user\_type — Ещё одно поле с выбором вариантов, которое указывает тип пользователя (например, студент или сотрудник). Это помогает структурировать роли пользователей и управлять доступом к различным частям системы в зависимости от их роли.
8. university\_name и workplace — Дополнительные поля, которые могут хранить информацию о месте учебы или работы пользователя. Эти поля опциональны и могут быть пустыми, если такие данные не указаны.

Эти поля составляют структуру таблицы **CustomUser** в базе данных. Каждая строка в этой таблице представляет отдельного пользователя системы.



*20 нетестовых записей (10 студентов, 10 сотрудников) в таблице* **CustomUser**

**Связи и управление пользователями**

Модель пользователя в Django не существует сама по себе, ей необходим механизм управления для создания, обновления и удаления пользователей. Для этого используется специальный класс **CustomUserManager**, который управляет созданием обычных пользователей и суперпользователей.

**CustomUserManager**

Класс **CustomUserManager** отвечает за создание новых записей в таблице пользователей. Внутри него реализованы два основных метода:

1. create\_user — Метод для создания обычного пользователя. Здесь важно отметить, что поле email является обязательным, так как оно используется в качестве основного идентификатора (вместо username). При создании нового пользователя его электронная почта нормализуется (приводится к стандартному виду), а пароль хэшируется с использованием метода set\_password. После этого пользователь сохраняется в базе данных.
2. create\_superuser — Метод для создания суперпользователя, который будет иметь расширенные права доступа (например, доступ к админке). В этом методе также проверяются флаги is\_staff и is\_superuser, чтобы гарантировать, что суперпользователь создаётся с правильными привилегиями. Этот метод особенно важен, поскольку суперпользователи управляют приложением и могут выполнять административные задачи.

**Записи в базе данных**

Каждая запись в базе данных представляет собой отдельного пользователя с уникальными характеристиками. Когда создаётся новый пользователь через метод create\_user, для него создаётся новая строка в таблице **CustomUser**. Эта строка будет содержать уникальный email, ФИО (отчество может быть не указано, также могут быть составные отчества и фамилии) статус активности, дату присоединения и другие данные. Если пользователь является суперпользователем, в его записи будут установлены флаги is\_staff и is\_superuser в True, что даёт ему дополнительные права в системе.

**Поле USERNAME\_FIELD и REQUIRED\_FIELDS**

В модели **CustomUser** используется поле USERNAME\_FIELD, которое указывает, какое поле будет использоваться в качестве идентификатора пользователя. В данном случае это поле email. Django по умолчанию использует поле username для аутентификации, но здесь эта логика изменена для использования электронной почты. Поле REQUIRED\_FIELDS оставлено пустым, что означает, что для создания пользователя достаточно указать только email и пароль.

**PermissionsMixin**

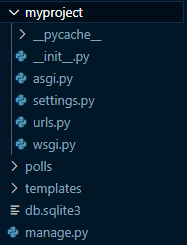
Класс PermissionsMixin, от которого наследуется модель **CustomUser**, добавляет стандартные поля и методы для работы с правами доступа пользователей. Это позволяет системе легко управлять авторизацией, назначать группы и права для пользователей, а также проверять их доступ к тем или иным частям системы.

1. **Django туториал**

Для выполнения данной работы автор прошёл туториал по Django, расположенный по ссылке [Django 4.2 Tutorial](https://docs.djangoproject.com/en/4.2/intro/overview/). Он представляет собой подробное пошаговое руководство, предназначенное для разработчиков, которые хотят научиться создавать веб-приложения с использованием фреймворка Django. Туториал охватывает базовые этапы разработки приложения, начиная с создания нового проекта и заканчивая управлением контентом через административную панель. В итоге прохождения туториала получается Django опросник, в котором можно голосовать за ответы на вопросы

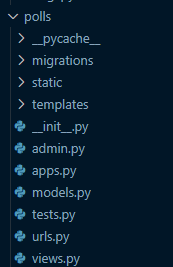
**Основные разделы туториала:**

1. **Создание нового проекта**: В этом разделе рассказывается, как установить Django и создать новый проект с помощью команды django-admin startproject. Это начальная точка любого проекта на Django, где объясняется структура директорий и файлов, создаваемых по умолчанию, таких как manage.py, settings.py, и urls.py.



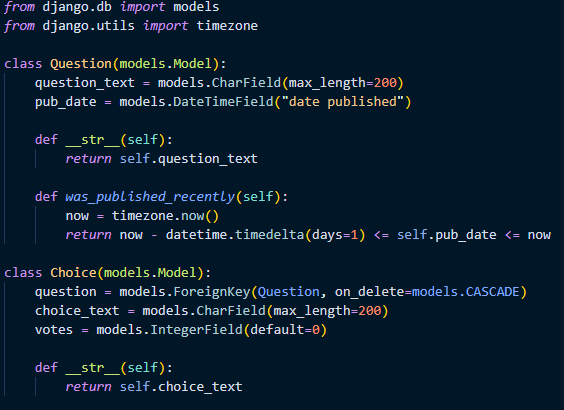
*Скриншот-доказательство выполненного этапа*

1. **Создание приложения**: Django имеет концепцию "приложений" — это самостоятельные модули внутри проекта, которые можно повторно использовать и подключать к другим проектам. Туториал показывает, как с помощью команды python manage.py startapp создать новое приложение. Также объясняется, как зарегистрировать приложение в основном проекте.



*Скриншот-доказательство выполненного этапа*

1. **Создание моделей**: Модели в Django — это классы, которые представляют структуру данных и автоматически преобразуются в таблицы базы данных. Туториал объясняет, как определить модели с полями для хранения данных, например, текстовые поля, даты или числа. Используя команду python manage.py makemigrations и python manage.py migrate, можно автоматически создать и обновить таблицы в базе данных.



*Скриншот-доказательство выполненного этапа*

1. **Работа с админкой**: Django имеет встроенную административную панель, которая позволяет управлять данными приложения через веб-интерфейс. Туториал показывает, как добавить модели в административный интерфейс и как использовать эту панель для создания, редактирования и удаления записей в базе данных.



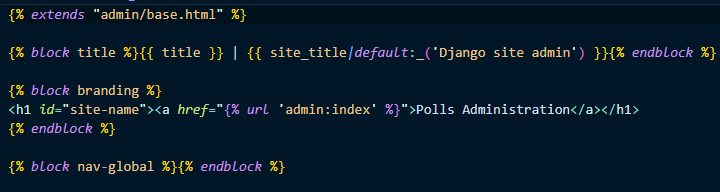
*Скриншот-доказательство выполненного этапа*

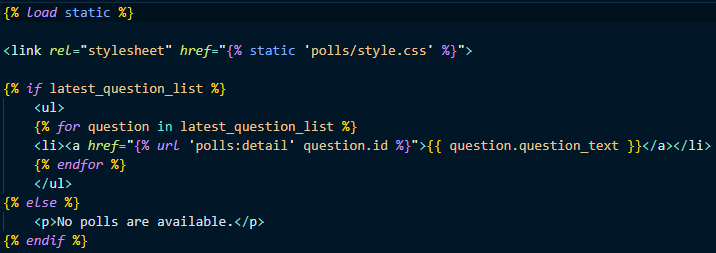
1. **Создание представлений (views)**: Представления (views) в Django — это функции или классы, которые обрабатывают запросы от пользователя и возвращают ответ, обычно в виде веб-страницы.



*Скриншот-доказательство выполненного этапа*

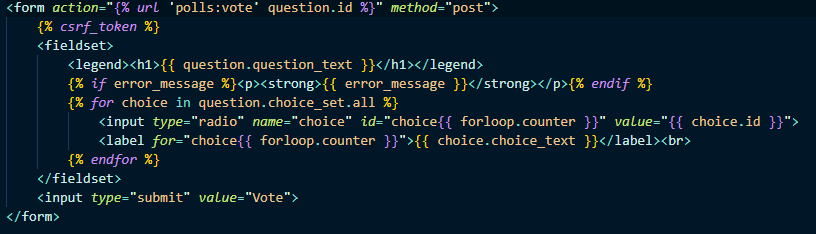
1. **Создание шаблонов (templates)**: Шаблоны (templates) используются для отображения данных на веб-страницах. Django использует собственный язык шаблонов (Django Template Language — DTL), который позволяет динамически генерировать HTML. Туториал объясняет, как создать HTML-шаблоны и передать данные из представлений в шаблоны для отображения на веб-страницах. В частности описывает создания базового интерфейса сайта: **base\_site.html**



**

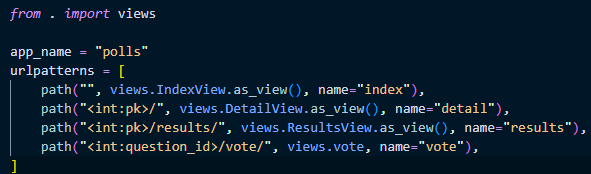
*Скриншот-доказательство выполненного этапа*

1. **Работа с формами**: В этом разделе рассказывается, как создавать и обрабатывать формы в Django. Формы позволяют пользователям взаимодействовать с сайтом, вводить данные, которые затем могут быть обработаны и сохранены в базе данных. Туториал объясняет, как Django автоматически создаёт формы на основе моделей и как обрабатывать данные, введённые пользователем.

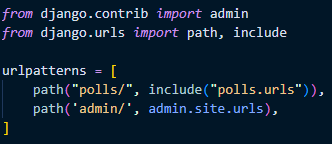
**

*Скриншот-доказательство выполненного этапа*

1. **Маршрутизация (URLs)**: Django использует систему маршрутизации для связывания URL с конкретными представлениями. Туториал учит, как настраивать URL-шаблоны с помощью функции path() и связывать их с представлениями, чтобы пользователи могли получать доступ к нужным страницам.

**

*Urls.py приложения polls*

**

*Urls.py всего проекта*

**5. CRUD: пример и реализация**

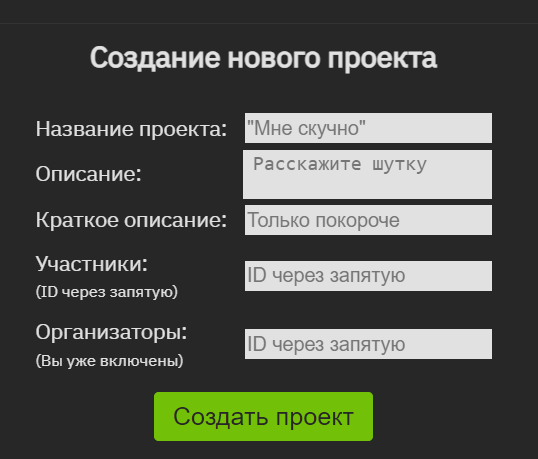
CRUD — это акроним, который обозначает четыре основных операции, выполняемых над данными в приложениях: **Create** (Создание), **Read** (Чтение), **Update** (Обновление) и **Delete** (Удаление). Эти операции являются базовыми для управления данными в базах данных и приложениях, работающих с любой формой информации. Каждая из этих операций соответствует действиям, которые пользователи или система могут выполнять с данными.

В контексте кода данного проекта для выполнения какой-либо операции с данными нужна связь models.py – views.py – urls.py. Модели базы данных уже были рассмотрены в этом документе, поэтому в этом разделе речь пойдёт от остальных двух файлах

Рассмотрим пример CRUD с соответствующим кодом:

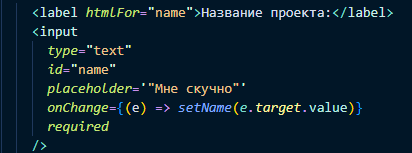
* **Create – создание проекта**

Для создания проекта пользователю необходимо заполнить форму, которая включает в себя подя «Название проекта», «Описание», «Краткое описание», а также поля «Участники» и «Организаторы»

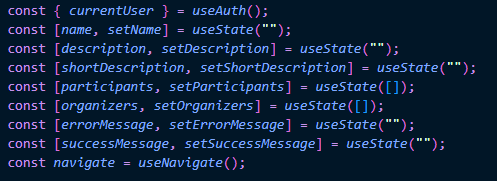


*Скриншот формы создания проекта*

Данные, которые вводит пользователь записываются в соотвутвующие переменные, которые инициализированные про помощи react «хука» useState.



*Поле для ввода имени проекта*

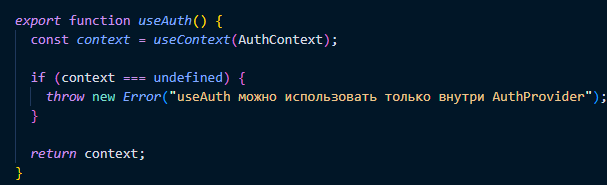
**

*Инициализация переменных для каждого поля формы*

Хук useState создаёт переменную и функцию, которая её обновляет. В качестве значений useState принимает начальное значение переменной. Как только изменяется значение переменной, все прочие переменные и html элементы обновляются («ререндерятся») с новым значением

Кроме useState мы также используем хук useNavigate от react-router-dom, расширения, позволяющего реализовать в приложении client-side routing. При таком подходе пользователь, переходя на сайт, сразу получает информцию о всех страницах сайта в JavaScript файле, который также «отрисовывает» страницы, когда пользователь переходит на соответствующий url. В таком случает начальная загрузка сайта будет дольше, однако переход по страницам будет быстрым и незаметным

Также мы используем кастомный хук useAuth, который возвращает id текущего пользователя



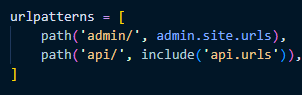
*Кастомный хук useAuth*

По нажатии кнопки «Создать проект» вызывается функция handleSubmit, которая собирает все данные в один объект и методом fetch отправляет по ссылке, настроенной в urls.py. В ответ на запрос пользователь получает JSON объект в котором храниться id ново созданного проекта. Далее при помощи useNavigate функцию перенаправляет пользователя на страницу созданного проекта

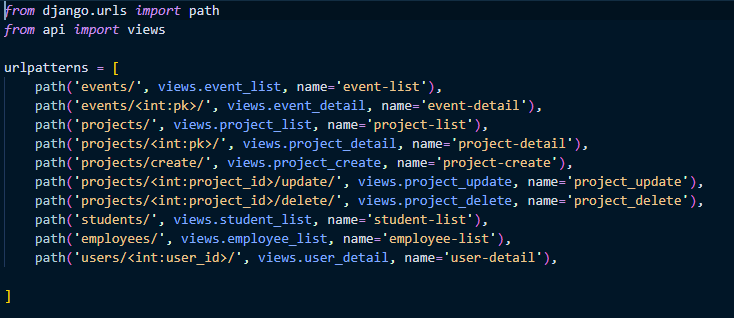


*Функция handleSubmit*

Ссылка, на которую функцию отправляет данные определена в urls.py Django приложения, названного api. Упоминая приложения, необходимо сказать, что серверная часть проекта поделена на приложения. Приложение api отвечает за работу с данными о проектах и мероприятиях, оно также определяет модели этих данных. Второе приложение, accounts, отвечает за работу с пользователями и их данными. Django проект, связывающий эти приложения, называется backend. Он хранит общий urls.py, в котором прописаны маршруты панели администратора и api.



*Urls.py всего Django проекта*

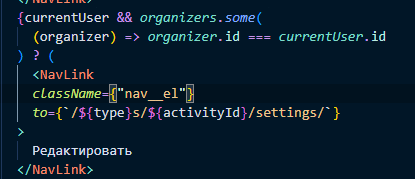


*Urls.py приложения api*

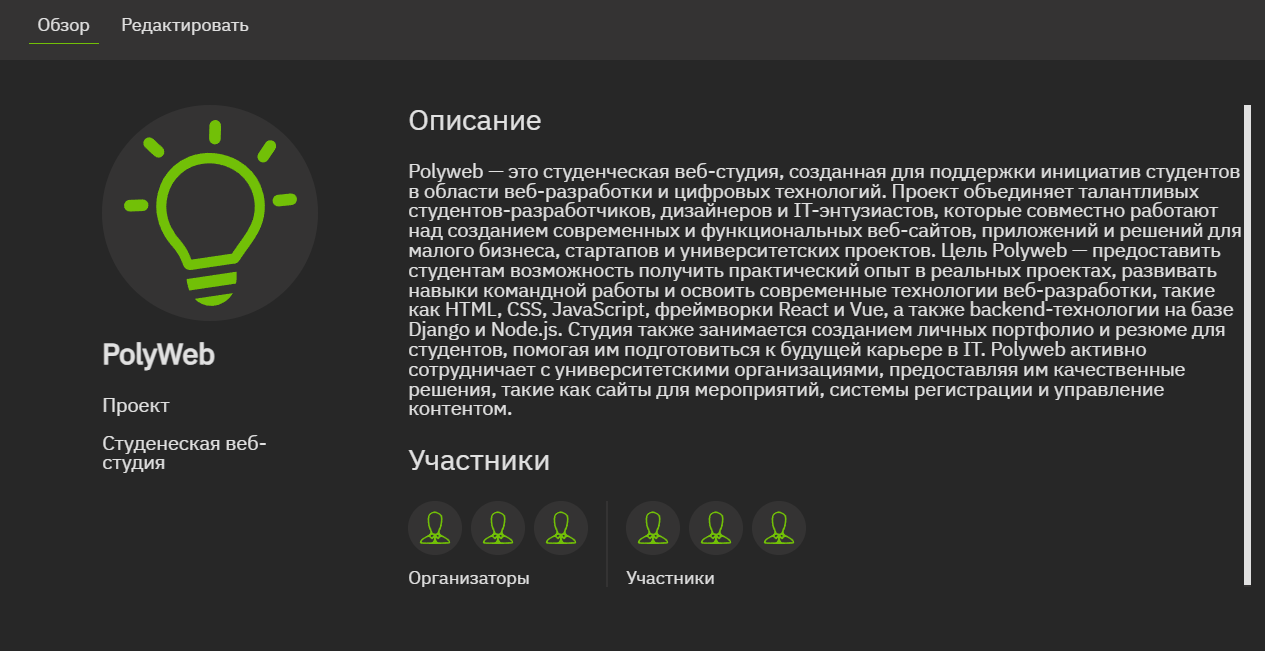
Данные отправленные по ссылке попадают в функцию файла views.py под названием *project\_create,* принимающую запрос как аргумент. Проверив метод запроса, Django парсит json объект, получает данные, указанные в форме и создаёт новую запись **Project**. Для полей участников и организаторов, Django из всех записей о пользователях выбирает те, которые соответствуют переданным id.

* **Read – чтение информации о проекте**

Страница проекта предоставляет пользователю информацию о названии, описании, типе активности (проект в данном случае), описании активности, а также об его участниках и организаторах. По нажатии на иконки пользователей, откроется страница с их профилями. На «шапке» странице располагаются 2 ссылки, на основную информацию об активности и на страницу редактирования. Последняя доступна только организаторам.



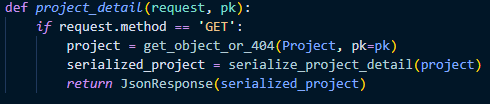
*Проверка на организатора*



*Страница проекта*

Данные об активности пользователь получает при помощи функции *getData,* которая запрашивает информацию у сервера. Пока данные не получены, вместо информации отображается слово «Загрузка…», *setIsLoading,* отвечает за эту функцию.

Стороне сервера, Django исполняет функцию *project\_detail,* описанную в views.py



*Функция project\_detail*

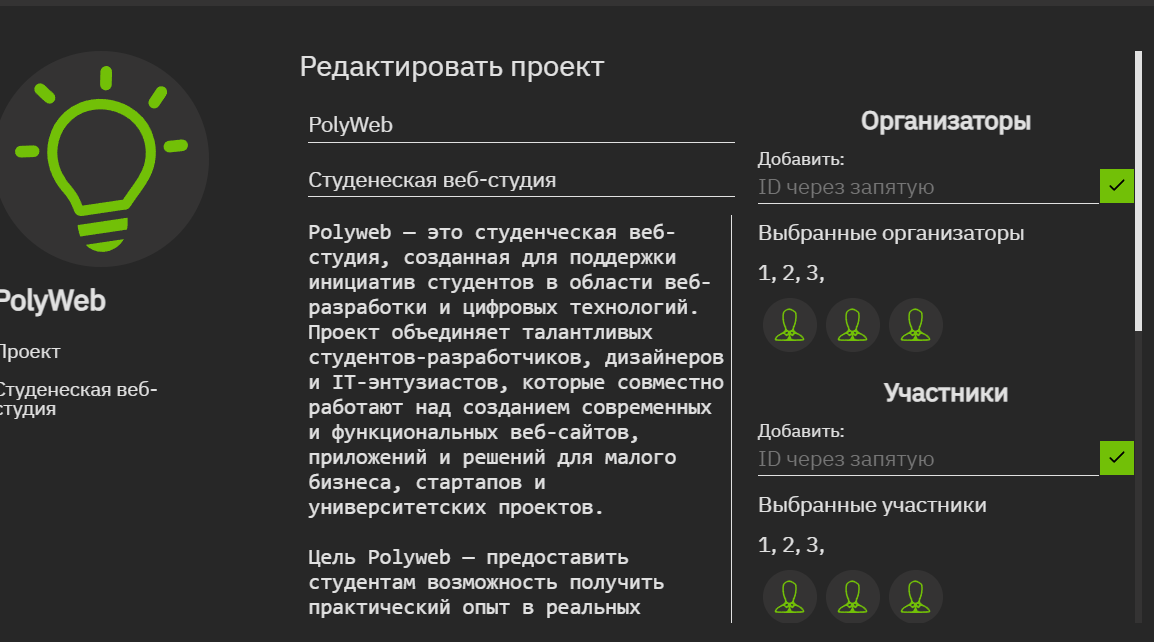


*url project\_detail*

Функция получает pk – primary key – из url и пытается найти запись Project с советующим pk. В случае успеха пользователю отправляется JSON объект с информацией о проекте, иначе возникает ошибка 404 – not found

* **Update -- обновление информации о проекте**

Перейдя на вкладку «Редактировать», организатор увидит форму с данными о проекте, а также список участников и других организаторов



*Вкладка «Редактировать»*

За получение данных ответственна вышеописанная функция *getData.* Реализация формы редактирования схож с реализацией формы создания проекта.

По нажатии на копку «Сохранить изменения», вызывается функция *handleSubmit,* отправляющая новые данные на сервер. В случае успешного изменения данных об активности, пользователя, спустя секунду, переадресовывает на страницу данной активности.

Данные отправляются на сервер, где Django вызывает функцию *project\_update.* Она схоже с вышерассмотренной функцией создания проекта.

До оглашения функции в коде использован декоратор @*csrf\_exempt,* отключающий проверку на Cross-Site Request Forgery (CSRF), которая требует отдельного скрытого поля с уникальным кодом, который в последствии проверяется на сервере. Функция возвращает JSON объект с id изменённого проекта



*Функция project\_update*



*Функция handleSubmit*

* **Delete -- удаление информации о проекте**

На этой же странице располагается кнопка удаления активности (проекта в данном случае). В качестве защиты от случайного нажатия, для удаления данных на копку нужно нажать 3 раза, при этом каждый раз текст изменяется, всё серьёзнее спрашивая намерения пользователя.



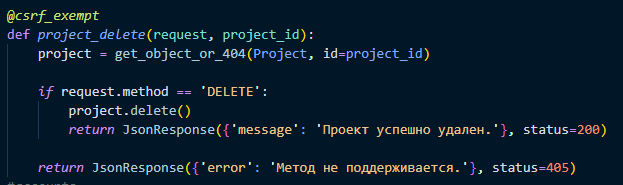
*Html кнопки удаления*



*Функционал «3 клика» по кнопке удаления*

На 3 нажатие вызывается функция *deleteProjectAPI,* отправляющая запрос на удаление активности. В случае успеха пользователя переадресует на страницу всех активностей данного типа (проектов в данном случае)

На сервере Django вызывает функцию *project\_delete,* получающую project\_id из url. Здесь также используется декоратор @*csrf\_exempt.* Для удаления записи вызывается метод delete().



*Функцию project\_delete*



*Функция deleteProjectAPI*

**6. Django панель администратора**

Панель администратора (или админка) — это специальный интерфейс, предназначенный для управления данными приложения, конфигурацией системы и пользователями. В backend-языках админка позволяет администраторам или владельцам системы управлять базами данных, контентом и пользовательскими правами без необходимости напрямую взаимодействовать с кодом или базой данных.

Админ-панели присутствуют в большинстве веб-фреймворков на backend-языках, таких как PHP (Laravel, Symfony), Ruby (Rails), и Python (Django)

**Преимущества админки Django**

Встроенная панель администратора Django обладает множеством преимуществ, которые делают её одной из самых удобных и мощных среди других фреймворков.

1. **Мгновенная генерация админки**:
   * Django предоставляет готовую панель администратора по умолчанию. Как только создаются модели данных, достаточно зарегистрировать их в админке с помощью нескольких строк кода, чтобы получить полностью рабочий интерфейс для управления этими данными.
2. **Автоматическая интеграция с моделями**:
   * Админка автоматически генерирует интерфейс на основе моделей, созданных в приложении. Поля модели автоматически отображаются в формах создания и редактирования объектов.
   * Админка поддерживает такие отношения между моделями, как ManyToMany и ForeignKey, и предоставляет средства для удобного управления связями через инлайны или фильтры.
3. **Поддержка кастомных интерфейсов**:
   * Хоть в данной работе это преимуществ осталось не реализованным, Django admin позволяет добавлять кастомные страницы и интерфейсы. Это полезно, если нужно расширить стандартные возможности админки, создавая специфические формы.
4. **Встроенная поддержка международных версий (интернационализация)**:
   * Django admin поддерживает интернационализацию, что позволяет адаптировать панель администратора для разных языков и регионов.

Логика работы панели администратора в Django реализована через файлы admin.py. Так как Django предоставляет базовый интерфейс панели администратора для наала работы достаточно добавить одну строку для каждой модели данных:

*admin.site.register(НазваниеМодели)*

Получить доступ к панели можно через сайт администратора, однако прежде нужно зарегистрировать данный URL:

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

Для кастомизации вида и удобства работы с данными существуют несколько иных способов регистрировать модели в панель администратора: декораторы и отдельный класс, отвечающий за внешний вид и представление данных. В данной работе используются оба варианта

Backend данного веб-сервиса состоит из 2 приложений, account и api, поэтому в проекте существуют 2 admin.py файла, соответственно отвечающих за эти приложения

**«Админка» api**

Этот файл admin.py отвечает за настройку админской панели для моделей Event и Project. Каждая из моделей имеет свои особенности отображения и функционала в админке, что делает работу с данными более удобной и организованной.

Для регистрации моделей в панели администратора используется декоратор @*admin*.*register*(НазваниеМодели). Ниже располагаются классы, отвечающие за отображение данных.



*«Админка» api*

В данном файле реализованы несколько настроек для улучшения отображения данных.

**list\_display**

* **Что делает**: определяет, какие поля модели будут отображаться в таблице списка объектов в админке.
* **Где используется**:
  + В классе EventAdmin указаны поля: name, start\_date, end\_date, address, а также кастомное поле short\_description\_preview.
  + В классе ProjectAdmin указываются поля name и short\_description\_preview.
* **Зачем используется**: Эта функция позволяет администратору видеть основные данные объектов модели на странице списка, что делает интерфейс удобным для быстрого просмотра и навигации по данным.

*list\_display = ('name', 'start\_date', 'end\_date', 'address', 'short\_description\_preview')*

**2. list\_filter**

* **Что делает**: Добавляет фильтры для удобной сортировки и фильтрации объектов по указанным полям.
* **Где используется**:
  + В EventAdmin фильтрация осуществляется по полям start\_date, end\_date, и organizers.
  + В ProjectAdmin фильтр есть только по полю organizers.
* **Зачем используется**: Это упрощает администратору фильтрацию объектов на основе дат начала/окончания или по организаторам, позволяя быстрее находить нужные данные.

*List\_filter = ('start\_date', 'end\_date', 'organizers')*

**3. inlines**

* **Что делает**: Добавляет инлайны для работы с отношениями ManyToMany. Это позволяет добавлять, удалять и редактировать связанные объекты прямо на странице редактирования основного объекта.
* **Где используется**:
  + В EventAdmin инлайны ParticipantsInline и OrganizersInline добавляют участников и организаторов событий.
  + В ProjectAdmin аналогично добавлены инлайны для участников и организаторов проектов.
* **Зачем используется**: Инлайны удобны для управления связанными объектами без необходимости перехода на отдельные страницы. Это ускоряет процесс редактирования данных.

*inlines = [ParticipantsInline, OrganizersInline]*

**4. date\_hierarchy**

* **Что делает**: Добавляет панель для навигации по объектам на основе дат. Администратор может быстро фильтровать объекты по годам, месяцам и дням.
* **Где используется**: В EventAdmin указана иерархия по дате начала события (start\_date).
* **Зачем используется**: Удобно при большом количестве событий, позволяет администратору быстро находить события, происходящие в определённый период.

*date\_hierarchy = 'start\_date'*

**5. @admin.display**

* **Что делает**: используется для настройки метода, который будет отображаться в админке как одно из полей модели.
* **Где используется**: В обоих классах (EventAdmin и ProjectAdmin) для отображения сокращённого описания через метод short\_description\_preview.
* **Зачем используется**: Позволяет выводить кастомные данные (например, только первые 50 символов описания), что помогает сэкономить место и улучшить визуальное представление информации.

*@admin.display(description='Short Description Preview')*

*def short\_description\_preview(self, obj):*

*return obj.short\_description[:50]*

**6. filter\_horizontal**

* **Что делает**: Позволяет отображать поля ManyToMany в виде горизонтального списка с двумя полями (выбранные и доступные объекты), что удобно при большом количестве связанных записей.
* **Где используется**: В обоих классах (EventAdmin и ProjectAdmin) для полей participants и organizers.
* **Зачем используется**: Это улучшает работу с полями ManyToMany, позволяя более удобно выбирать участников и организаторов.

*filter\_horizontal = ('participants', 'organizers')*

**7. list\_display\_links**

* **Что делает**: Определяет, по каким полям можно кликать, чтобы перейти на страницу редактирования объекта.
* **Где используется**:
  + В EventAdmin поля name и start\_date сделаны кликабельными.
  + В ProjectAdmin — поле name.
* **Зачем используется**: Упрощает навигацию по объектам, позволяя администраторам быстрее перейти к редактированию нужного объекта.

*list\_display\_links = ('name', 'start\_date')*

**8. raw\_id\_fields**

* **Что делает**: Заменяет выпадающие списки для выбора связанных объектов (например, участников и организаторов) на поля ввода, куда можно ввести ID объекта.
* **Где используется**: В обоих классах для полей participants и organizers.
* **Зачем используется**: Это улучшает производительность при большом количестве объектов, так как не загружается весь список, а вводится конкретный ID.

*raw\_id\_fields = ('participants', 'organizers')*

**9. readonly\_fields**

* **Что делает**: Делает поля доступными только для чтения (их нельзя редактировать в админке).
* **Где используется**:
  + В EventAdmin поле address помечено как только для чтения.
  + В ProjectAdmin — поле description.
* **Зачем используется**: Поля, которые не должны быть изменены через админку, можно сделать только для чтения, что защищает важные данные от случайного изменения.

*readonly\_fields = ('address',)*

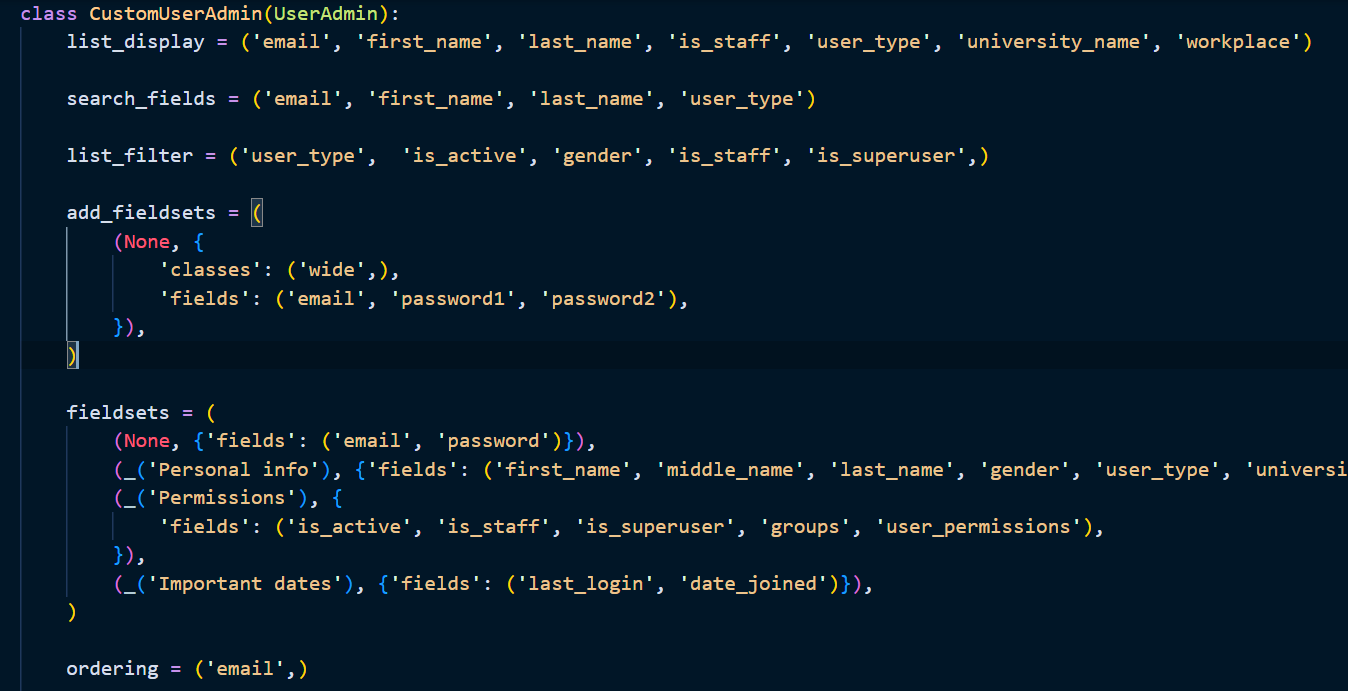
**10. search\_fields**

* **Что делает**: Определяет поля, по которым можно выполнять поиск объектов.
* **Где используется**:
  + В EventAdmin поиск осуществляется по полям name, description, и short\_description.
  + В ProjectAdmin — по тем же полям.
* **Зачем используется**: Это помогает быстро находить объекты на основе ключевых слов в их именах или описаниях.

*search\_fields = ('name', 'description', 'short\_description')*

**«Админка» accounts**

Этот admin.py файл отвечает за регистрацию и настройку кастомной модели пользователя CustomUser в административной панели Django. Здесь используется класс CustomUserAdmin, который расширяет стандартный UserAdmin для кастомной модели пользователя.



*«Админка» accounts*

**Регистрация модели в админке**

Регистрация модели в админке позволяет отображать её в интерфейсе администратора. Для этого используется метод admin.site.register, который связывает модель с её административным классом CustomUserAdmin.

**CustomUserAdmin**

Класс CustomUserAdmin наследует стандартный класс UserAdmin и настраивает отображение, поиск, фильтрацию и поля модели пользователя. Благодаря этому администратор может эффективно управлять пользователями через удобный интерфейс.

* Для отображения данные используется вышеописанный метод list\_display.
* search\_fields определяет, по каким полям можно искать пользователей.
* Поля фильтрации определяет list\_filter.
* add\_fieldsets определяет, какие поля будут отображаться при создании нового пользователя в админке.
* fieldsets определяет структуру и порядок полей на странице редактирования пользователя.
* ordering отвечает за порядок сортировки записей