**Глава 4(Создание представления входа в систему)**

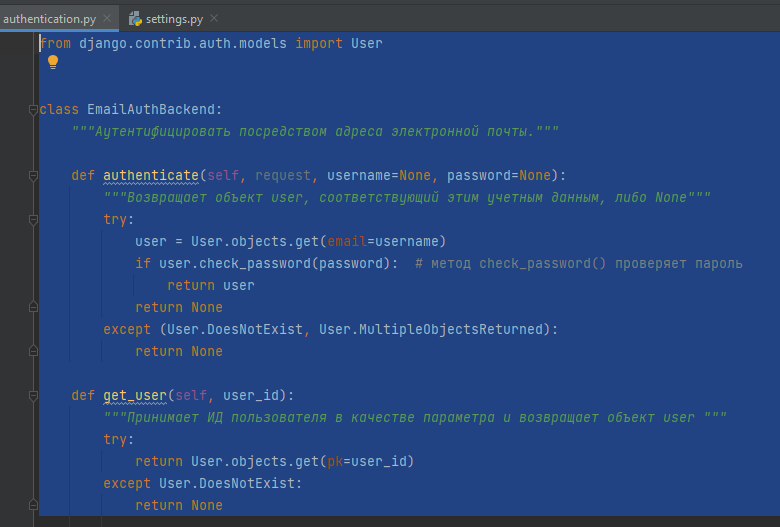
**authenticate()** = **Аутентифицирует** пользователя по базе данных. Указанный метод принимает объект request, параметры username и password и возвращает объект User, если пользователь был успешно аутентифицирован, либо **None** в противном случае.

**is\_active =** Проверяет активен ли пользователь

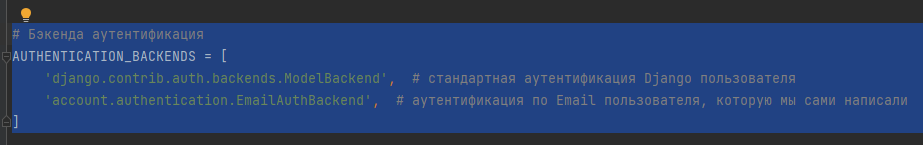
**login()** = Задает пользователя в текущем сеансе для входа пользователя в сайт. ВХОД выполняется вызовом функции **login**. Запрос должен быть представлен экземпляром класса HttpRequest, а пользователь, от имени которого выполняется вход, — записью модели User.

**Глава 4(Разработка конкретно-прикладного бэкенда аутентификации)**

Мы создали **дополнительный** бэкенд **аутентификацию**, для этого мы создали файл **authentication.py** и в нем прописали класс аутентификации



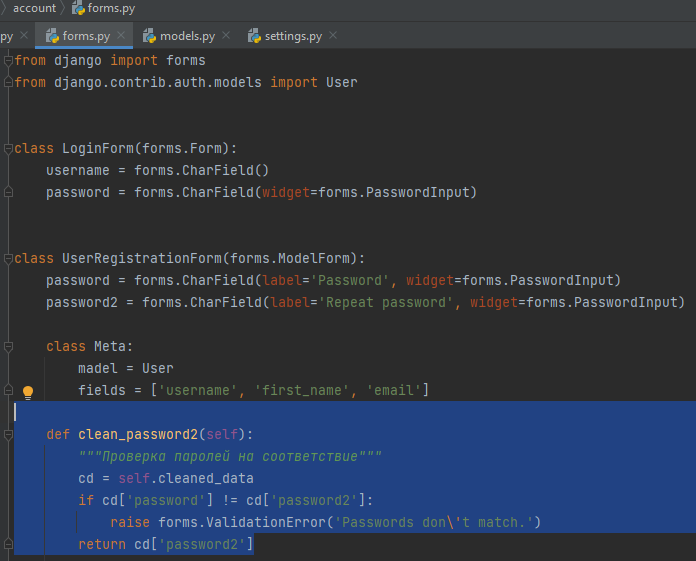
После в настройках проекта добавили собственную аутентификацию после встроенного, и теперь пользователь может аутентифицироваться как по **имени**, так и по **email**



**Глава 4(**Регистрация пользователя**)**

В форме если нужно проверить валидацию отдельного поля можно использовать метод с началом **clean\_<название поля>**. Такие методы исполняются, когда форма проходит валидацию при вызове метода is\_valid(). А если хотите проверить/валидировать всю форму можно использовать метод **clean()**

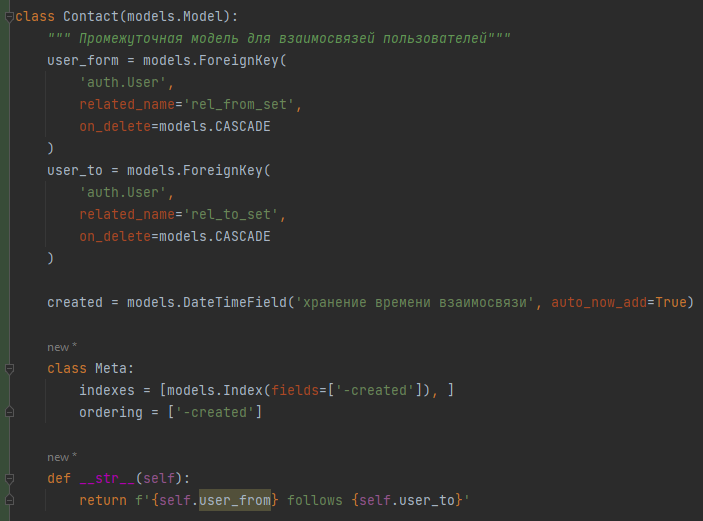
Пример:



**Глава 7(Формирование взаимосвязей многие-ко-многим с промежуточной моделью)**

Создание промежуточной модели необходимо, когда требуется хранить дополнительную информацию о взаимосвязи, например дату создания взаимосвязи или поле, описывающее природу взаимосвязи.

Фото промежуточной модели:



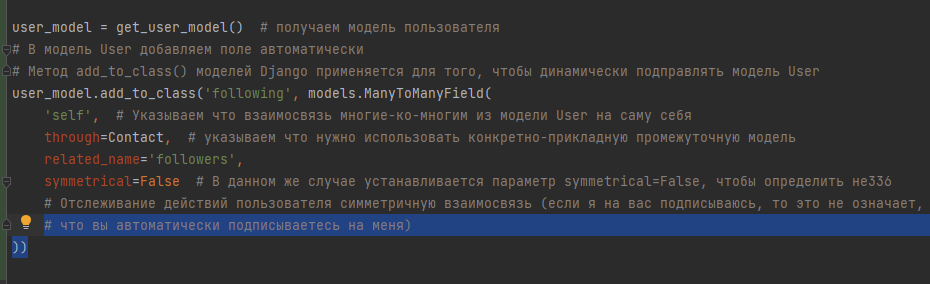
• user\_from: внешний ключ (ForeignKey) для пользователя, который создает взаимосвязь;

• user\_to: внешний ключ (ForeignKey) для пользователя, на которого есть подписка;

• created: поле DateTimeField с параметром auto\_now\_add=True для хранения времени создания взаимосвязи.

Мы в модель пользователей (User) добавили поле **динамическим** способом.

Фото примера:



**get\_user\_model()** = Извлекаем модель пользователя

**add\_to\_class()** = моделей Django применяется для того, чтобы динамически подправлять модель User

**'self'** = Указываем что взаимосвязь многие-ко-многим из модели User на саму себя

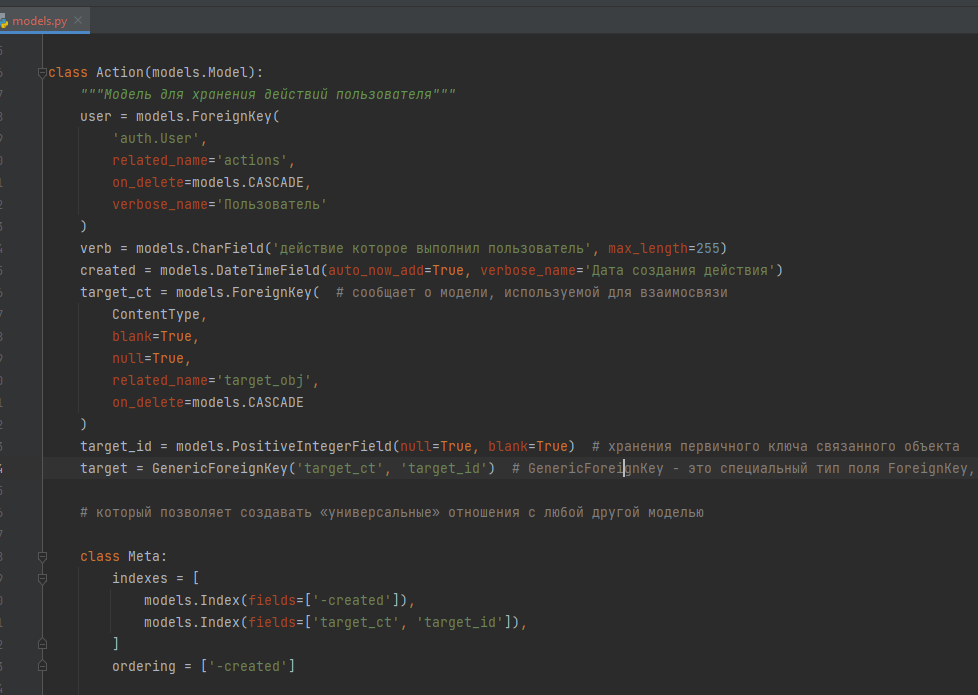
**through=Contact** = указываем что нужно использовать конкретно-прикладную промежуточную модель

**symmetrical=False** = При определении поля ManyToManyField в модели, создавая взаимосвязь с самой моделью, Django навязывает взаимосвязи симметричность. В данном же случае устанавливается параметр **symmetrical=False**, чтобы определить **несимметричную взаимосвязь** (если я на вас подписываюсь, то это не означает, что вы автоматически подписываетесь на меня).

**Глава 7 (Применение фреймворка contenttypes)**

**Contenttypes =** отслеживать все установленные в проекте модели и предоставляет типовой интерфейс взаимодействия с этими моделями. Если нужно возможность подключатся к разным моделям, используй contenttypes

Фото примера:



**target\_ct** = поле ForeignKey, указывающее на модель **ContentType**;

**target\_id** = **PositiveIntegerField** для хранения первичного ключа связанного объекта;

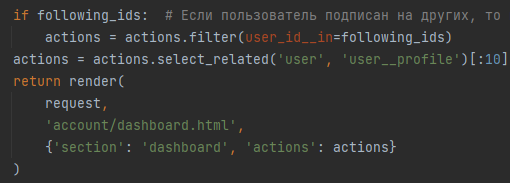
**target** = поле **GenericForeignKey** для связанного объекта на основе комбинации двух предыдущих полей.

**Глава 7 (Оптимизация наборов запросов, предусматривающих связанные объекты)**

**Применение метода select\_related.**

Используется для оптимизации работы с базой данных при работе с внешними ключами **один-ко-многим**. Он возвращает QuerySet, который будет «следовать» за отношениями внешнего ключа, выбирая дополнительные данные связанных объектов при выполнении своего запроса. Метод select\_related предназначен для полей **ForeignKey** и **OneToOne**.

Фото:



В этом примере **select\_related** используется для оптимизации запроса к базе данных. Он указывает Django выбрать связанные данные для полей **user** и **user\_\_profile** при выполнении запроса к базе данных. Это означает, что при итерации по объектам actions, доступ к связанным данным пользователя и профиля пользователя не потребует дополнительных запросов к базе данных.

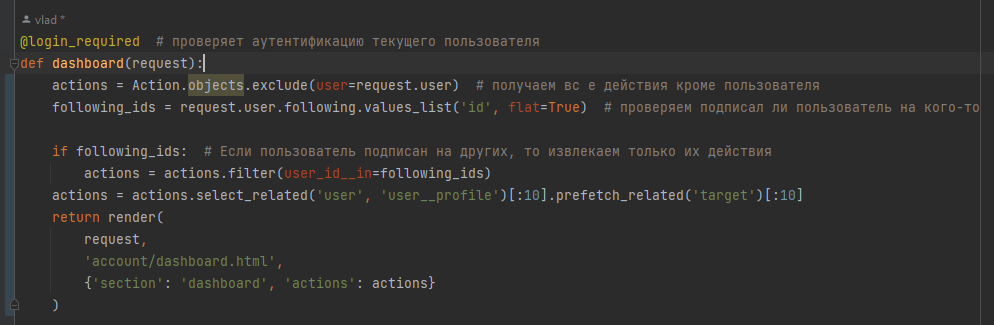
Кроме того, в этом примере используется срез [:10], чтобы ограничить количество возвращаемых объектов до 10. Это означает, что запрос вернет только первые 10 объектов из результирующего набора.

**user\_\_profile** - это обозначение связи между двумя моделями в Django. Двойное подчеркивание \_\_ используется для указания связи между моделями. В данном случае user\_\_profile означает, что мы хотим получить доступ к связанному объекту profile, который связан с объектом user. Эта связь определяется в модели User, где указывается, что у каждого пользователя есть профиль

**Применение метода prefetch\_related()**

**prefetch\_related** - это метод, который используется для оптимизации запросов к базе данных в Django. Он работает аналогично методу **select\_related**, но используется для связей “**многие-ко-многим**” и “ **многие-к-одному** ”. **prefetch\_related** выполняет отдельный запрос для получения связанных объектов и затем “склеивает” их с основными объектами в Python. Это позволяет уменьшить количество запросов к базе данных при обращении к связанным объектам.

Фото:



**Глава 7 (Использование сигналов  
для денормализации количественных  
данных)**

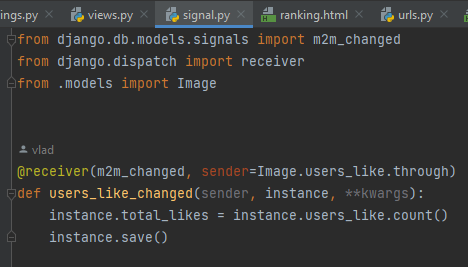
**Денормализация** - это процесс, который используется для ускорения работы базы данных. Она достигается за счет добавления дополнительных данных, которые могут быть избыточными. Это может быть полезно в случаях, когда требуется быстрый доступ к данным.

**Добавление избыточных данных** означает, что в базе данных хранится больше информации, чем необходимо для ее работы. Это может быть полезно, потому что позволяет ускорить доступ к данным, так как информация уже находится в базе данных и не требует дополнительных запросов. Однако это также может привести к увеличению размера базы данных и замедлению операций записи.

**Денормализация** – по-простому это лишние поля в модели, что бы не обращаться к другим моделям.

Создали файл signal.py и написали сигнал.

Фото примера:



**@receiver(m2m\_changed, sender=Image.users\_like.through)** - это декоратор, который указывает, что функция users\_like\_changed является приемником сигнала m2m\_changed. Этот сигнал срабатывает при изменении связи многие-ко-многим (many-to-many, m2m) между моделями. Аргумент sender указывает на промежуточную модель, которая управляет связью между моделями Image и users\_like.

**def users\_like\_changed(sender, instance, \*\*kwargs)** Это определение функции users\_like\_changed, которая вызывается при срабатывании сигнала m2m\_changed. Функция принимает несколько аргументов:

**sender** - это промежуточная модель, которая управляет связью между моделями Image и users\_like.

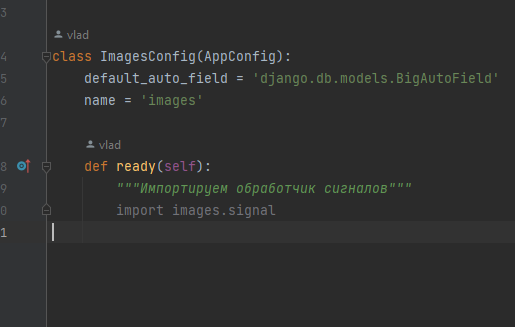
instance - это экземпляр модели Image, связь которого с моделью users\_like была изменена.

**\*\*kwargs** - это дополнительные аргументы ключевого слова, которые передаются в функцию.

**instance.total\_likes = instance.users\_like.count** в этой строке кода вычисляется значение поля total\_likes экземпляра instance. Это делается путем подсчета количества связанных объектов

**users\_like.instance.save** вэтой строке кода обновленное значение поля total\_likes сохраняется в базе данных.

Для регистрации сигнала мы в файле **apps.py** прописываем:



метод **ready**, который вызывается Django, когда приложение готово к использованию. В этом методе импортируются обработчики сигналов из модуля **images.signals**. Это необходимо для того, чтобы обработчики сигналов были зарегистрированы и могли срабатывать при соответствующих событиях.