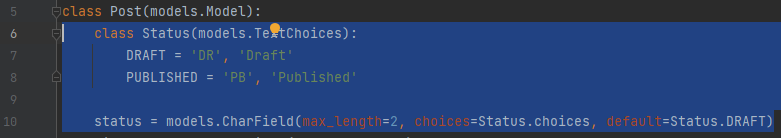
**Глава 1(Добавление поля статуса)**

Для выбора значение мы определили перечисляемый класс путем подклассирования класса.



Для получения имеющихся вариантов: **Post.Status.choices**

Вывод: [('DF', 'Draft'), ('PB', 'Published')]

Для получения удобочитаемые имена: **Post.Status.labels**

Вывод: ['Draft', 'Published']

Для получения фактических значений: **Post.Status.values**

Вывод: ['DF', 'PB']

Для получения конкретного значения элемента: **Post.Status.DRAFT.value**

Вывод: 'DR'

Для получения конкретного значения элемента: **Post.Status.DRAFT.name**

Вывод: 'DRAFT'

**Глава 1(Создания сайта администрирования для моделей)**

Настройка админки:

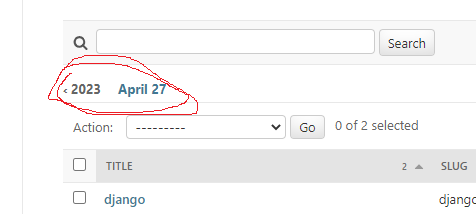
**list\_filter** – активирует боковую панель фильтрации

**prepopulated\_fields** - автоматическое заполнение слага

**raw\_id\_fields** - это список полей, которые вы хотели бы превратить в Input виджет для a ForeignKey или ManyToManyField

**date\_hierarchy** - быстрая фильтрация по датам

Фото пример:



**Глава 1** (**Когда вычисляются наборы запросов QuerySet )**

Наборы запросов QuerySet вычисляются только в следующих ниже случаях:

**•** при первом их прокручивании в цикле;

**•** при их нарезке, например Post.objects.all()[:3];

**•** при их консервации в поток байтов или кешировании;

**•** при вызове на них функций repr() или len();

**•** при вызове на них функции list() в явной форме;

**•** при их проверке в операциях bool(), or, and или if.

**Глава 1 (Добавление шаблонов URL-адресов представлений)**

**namespace** – может пригодиться в случае, если в проекте используется несколько экземпляров одного и того же приложения, которые манипулируют разными данными.

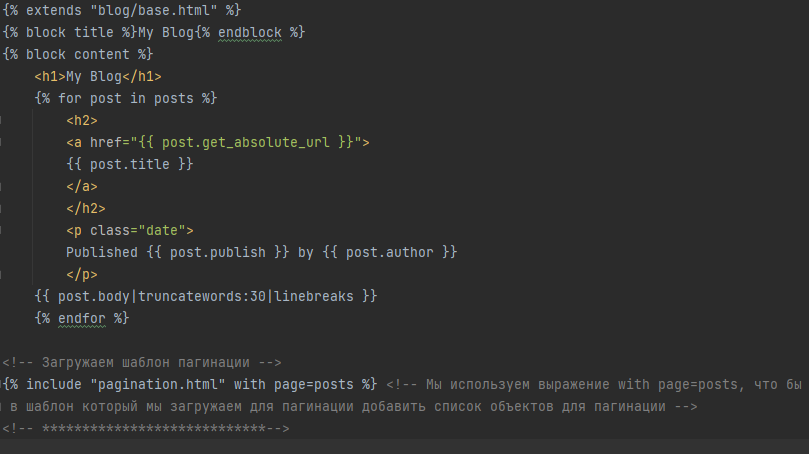
**Глава 2(Использование канонических URL-адресов для моделей)**

Канонический URL-адрес – это предпочтительный URL адрес ресурса,

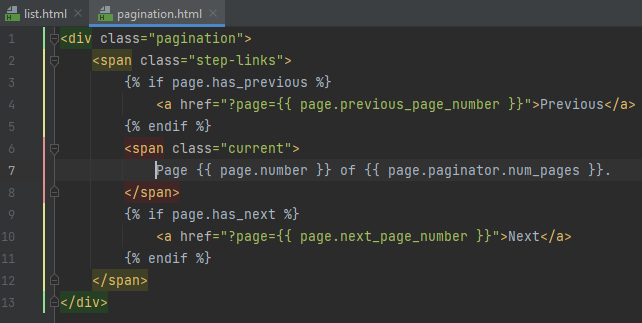
Простыми словами лучше использую метод **get\_absolute\_url** в шаблонах

**Глава 2(Создание шаблона постраничной разбивки)**

Мы используем выражение with page=posts, что бы передать в шаблон (который импортируем для пагинации ) список объектов которые нужно разбить на страницах



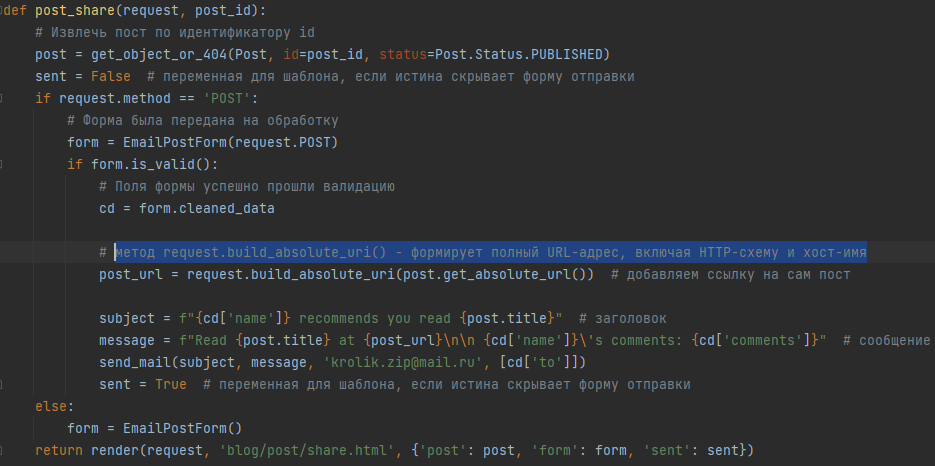
Пример шаблона который я импортирую для пагинации pagination.html



**Глава 2(Отправка электронных писем в представлениях)**

метод **request.build\_absolute\_uri()** - формирует полный URL-адрес, включая HTTP-схему и хост-имя.

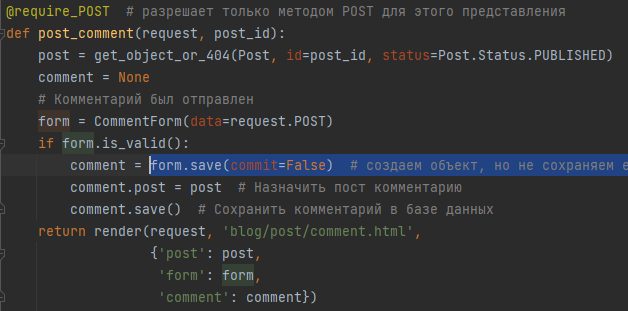
Пример кода:



**Глава 2(Оперирование формами ModelForm в представлениях)**

Декоратор **require\_POST**  - разрешает только методом POST **form.save(commit=False)** - создаем объект в оперативной памяти, но не сохраняем его в базе данных, чтобы модифицироваться его данные (добавлять/изменять данные)

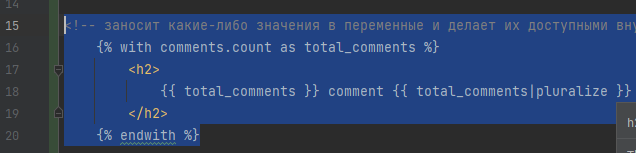
Фото пример:



**Глава 2(Добавление комментариев в шаблон детальной информации о посте)**

**with** . . . **endwith**— заносит какие-либо значения в переменные и делает их доступными внутри своего содержимого. Может использоваться для временного сохранения в переменных результатов каких-либо вычислений (например, полученных при обращении к методу класса) — чтобы потом не выполнять эти вычисления повторно.

Пример:



**Глава 2(Создание шаблонов комментарной формы)**

<form action="{% url 'blog:post\_comment' post.id %}" method="post">

В теге form в **action** – мы указываем url на который, будет отправлена форма.

**Глава 3(Добавление функциональности тегирования)**

**django-taggit** – это приспособленное для реиспользования приложение,

которое в первую очередь предлагает модель **Tag** и менеджер для удобного

добавления тегов в любую модель. Исходный код приложения доступен для

просмотра на странице https://github.com/jazzband/django-taggit.

**Глава 3(**Извлечение постов по сходству)

Получаем список тегов которые есть в посте и используем один из методов QuerySet **values\_list()**Ему передается параметр **flat=True**, чтобы получить одиночные значения, такие как [1, 2, 3, ...], а не одноэлементные кортежи, такие как [(1,), (2,), (3,) ...]

Фото пример:



Отмечу основные методы QuerySet

**filter():** фильтрует набор данных и возвращает выборку данных, которая соответствует фильтру, также в виде объекта QuerySet

**exclude():** фильтрует набор данных и возвращает выборку, которая НЕ соответствует фильтру, в виде объекта QuerySet

**order\_by():** сортирует набор данных и возвращает отсортированную выборку в виде объекта QuerySet

**distinct():** возвращает объект QuerySet, который содержит только уникальные объекты

**values():** возвращает данные объектов в виде словаря

**values\_list():** возвращает данные объектов в виде списка

**none():** возвращает пустой набор данных в виде объекта EmptyQuerySet (класс производный от QuerySet)

**all():** возвращает объект QuerySet, который содержит все объекты из таблицы

**union():** объединяет два объекта QuerySet и возвращает объединенный объект QuerySet

**intersection():** возвращает пересечение двух объектов QuerySet в виде другого QuerySet

**difference():** возвращает разность двух объектов QuerySet в виде другого QuerySet

**raw():** выполняет переданный в качестве параметра SQL-запрос

**get():** возвращает один объект модели

**create():** создает один объект модели

**get\_or\_create():** возвращает один объект модели по указанным параметрам, а если он отсутствует, то создает его

**update\_or\_create():** обновляет один объект модели по указанным параметрам, а если он отсутствует, то создает его

**bulk\_create():** создает набор объектов модели

**save():** сохраняет один объект модели (если он ранее отсутствовал в БД, то добавляет его, есть присутствовал, то обновляет его)

**bulk\_update():** обновляет набор объектов

**count():** возвращает количество объектов в наборе

**in\_bulk():** возвращает словарь объектов, которые имеют определенные идентификаторы

**latest():** возвращает объект с самыми последними изменениями

**earliest():** возвращает объект с самыми ранними изменениями

**first():** возвращает первый объект набора

**last():** возвращает последний объект набора

**aggregate():** выполняет агрегатную функцию (нахождение суммы, среднего, минимального, максимального значения и т.д.) и возвращает результат в виде числа

**exists():** проверяет, есть ли в наборе как минимум один объект. Если есть, то возвращается True, иначе возвращается False

**contains():** проверяет, есть ли в наборе определенный объект. Если есть, то возвращается True, иначе возвращается False

**update():** обновляет один объект модели

**delete():** удаляет один объект модели

В этой главе мы также создали “Виртуальное поле модели ” так сказать. Что бы получить посты с подобными тегами и отсортировать их по количеству совпадающих тегов мы использовали метод **annotate().**

**annotate** - В джанго это инструмент добавления виртуального поля в ответ от базы.

Для подсчета совпадающих тегов, что бы на основе количества тегов сортировать их, мы использовали метод **Count. Count** — вычисляет число записей.

Фото пример:



**Глава 3(Добавление карты сайта)**

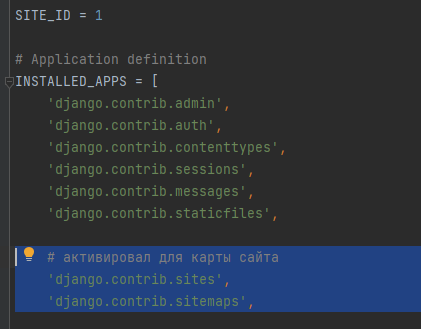
Django идет в комплекте с фреймворком карт сайтов, который позволяет

динамически создавать карты для своего сайта. Карта сайта – это XML-файл,

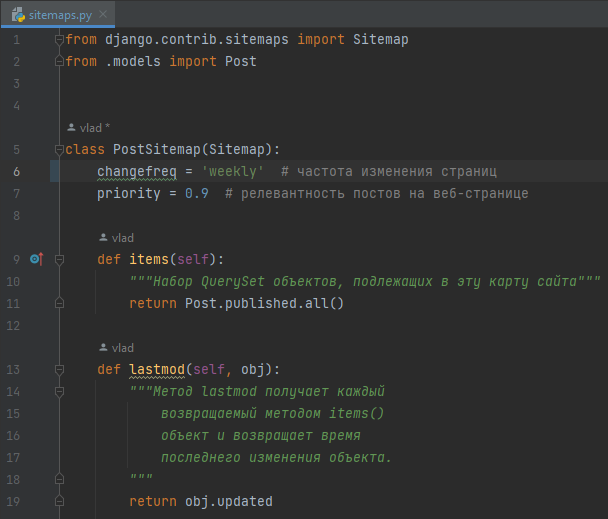
который сообщает поисковым системам о страницах веб-сайта, их релевантность и частоту их обновления. Использование карты сделает сайт **более заметным в рейтинге поисковых систем**, поскольку она помогает поисковым роботам индексировать содержимое сайта.

Ссылка на документацию: <https://docs.djangoproject.com/en/4.1/ref/contrib/sitemaps/>

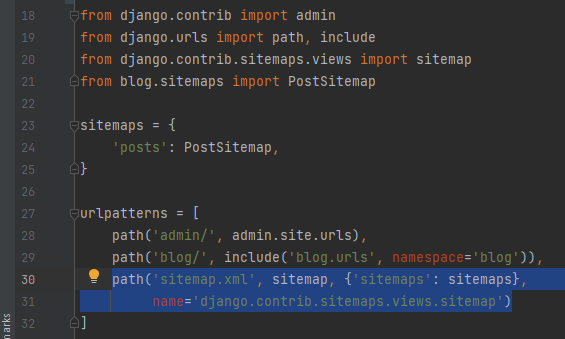
Для установки фреймворка карт сайтов необходимо активировать приложения sites и sitemap в своем проекте и определите новый настроечный параметр для ИД сайта.



После проводим миграцию, и создаем отдельный файл для карты сайты.



Последнее это создать для него url-адрес.



**Глава 3(Выгрузка существующих данных)**

Команда **dumpdata** выгружает данные из базы данных в стандартный вывод,

по умолчанию сериализованный в формате JSON. Результирующая структура

данных включает информацию о модели и ее полях, позволяя Django загружать ее в базу данных.

**python manage.py dumpdata --indent=2 --output=mysite\_data.json**

Если при выполнении команды вы получаете ошибку кодировки, то включите флаг-Xutf8, как показано ниже, чтобы активировать режим Python UTF-8:

**python -Xutf8 manage.py dumpdata --indent=2 --output=mysite\_data.json**

Выполните следующую ниже команду, чтобы **загрузить** данные в базу данных PostgreSQL:

python manage.py loaddata mysite\_data.json – это команда выдавала ошибки

**При попытки мигрировать с одной базы на другую выбивало ошибку, помогла команда**

python manage.py loaddata --exclude auth.permission --exclude contenttypes back.json

**Глава 3(Разработка представления поиска)**

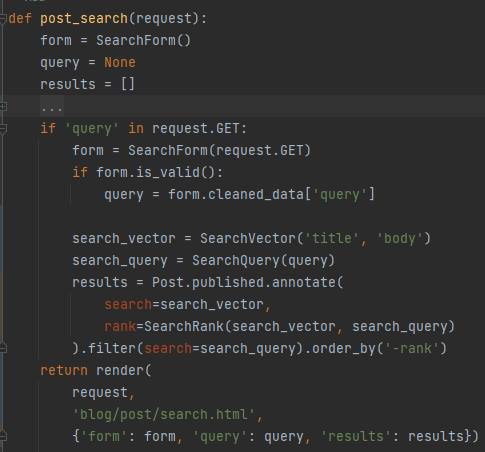
Что бы выполнить поиск по нескольким полям. В этом случае необходимо определить объект SearchVector:

**Post.published.annotate(search=SearchVector('title', 'body'),).filter(search=query)**

**Глава 3(Выделение основ слов и ранжирование результатов)**

**SearchQuery**, чтобы транслировать термины в объект поискового запроса

**SearchRank**, что бы ранжировать, упорядочить результаты на основе повторения терминов запроса



Объекты **SearchVector** **и SearchQuery** можно настроить под исполнение процедур выделения основ слов и удаления стоп-слов на любом языке. Для того чтобы использовать другую конфигурацию поиска, в SearchVector и SearchQuery передается атрибут config:

**search\_vector = SearchVector('title', 'body', config='spanish')**

**search\_query = SearchQuery(query, config='spanish')**

Влияние конкретных векторов можно усиливать таким образом, чтобы им

придавался больший вес при упорядочивании результатов по релевантности.

search\_vector = SearchVector('title', config='russian', **weight='A') +** SearchVector('body', config='russian'**, weight='B')**

В приведенном выше исходном коде к векторам поиска, сформированным

с использованием полей title и body, применяются разные веса. По умолчанию веса таковы: D, C, B и A, и они относятся соответственно к числам 0.1, 0.2, 0.4 и 1.0.

**Глава 3(Поиск по триграммному сходству)**

Фото пример:

