

Стажировка

«Веб-приложение для публикации постов – полнотекстовый поиск через postgres FTS»

Выполнил: Ноздряков Богдан Валериевич

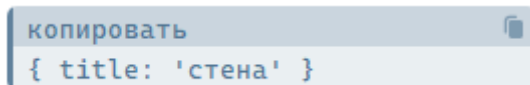
Проверил: Пармузин Александр Игоревич

Санкт-Петербург

2024

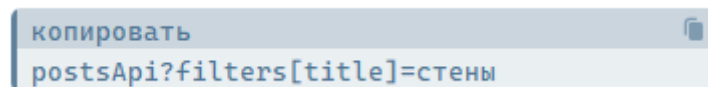
Условие:

Реализовать полнотекстовый поиск так, чтобы при наличии поста:



```
{ title: 'стена' }
```

Можно было осуществить поиск таким образом и найти этот пост:



```
postsApi?filters[title]=стены
```

Реализация через нативные средства на postgres – postgres FTS.

Анализ:

Полнотекстовый поиск (FTS):

-Перед тем, как разбираться что такое полнотекстовый поиск, важно разобрать следующие термины:

Документ – это некая единица обработки, например, статья, страница, или запись в базе данных, содержащая текстовые поля. Система поиска текста должна уметь разбирать документы и сохранять связи лексем (ключевых слов) с содержащим их документом. Впоследствии эти связи могут использоваться для поиска документов с заданными ключевыми словами.

Запрос – это текст, который мы ищем в документе. Может состоять из одного, или нескольких слов. Также может содержать логические операторы для более гибкого управления поиском.

Найденные документы соответствуют запросу. Однако понятие соответствия – очень размыто. Соответствие подразумевает, что в документе есть слова из теста запроса. Но также может учитываться частота повторения или то, на сколько близко стоят искомые слова друг к другу в тексте.

Стоп-слово – это такое слово, которое встречается во всех документах, и нет смысла искать по данному слову (предлоги, союзы, местоимения).

-Для того, чтобы реализовать механизм полнотекстового поиска, используя Postgres – существует два способа:

- Использование поисковых движков (Solr, elastic, Sphinx)
- Использование нативных средств

В текущей реализации – нужно реализовать полнотекстовый поиск, используя нативные средства Postgres. Из-за этого, мы не будем здесь разбирать способ через поисковые движки, а сосредоточимся на нативном Postgres.

-Итак, **полнотекстовый поиск** (или просто поиск текста) – это возможность находить документы на естественном языке, соответствующие запросу, и, возможно, дополнительно сортировать их по релевантности для этого запроса.

Сразу нужно пояснить, что возможность находить документы на естественном языке – имеется в виду, что поиск осуществляется с использованием обычных, повседневных выражений и фраз, которые люди используют в общении, а не специфических кодов или команд. Это отличает полнотекстовый поиск от других видов поиска, где могут потребоваться специальные запросы или синтаксис. Например, вместо использования сложных запросов на языке SQL для поиска информации в базе данных, пользователь может просто ввести фразу или слово на своем родном языке, и система найдет соответствующие документы или записи.

Сортировать по релевантности – имеется в виду, что найденные документы или записи будут упорядочены в соответствии с тем, насколько хорошо они соответствуют поисковому запросу пользователя. Релевантность определяется на основе различных факторов, таких как частота вхождения поисковых слов в документ, их расположение (например, наличие в заголовке или начале документа), а также другие параметры, которые могут учитываться поисковой системой или базой данных. Таким образом, документы, которые наиболее точно соответствуют запросу пользователя, будут отображаться первыми в результатах поиска.

-То есть, когда пользователь ищет товары в интернет-магазине, статьи в блоге и т.д. по текстовому запросу, как, например, в браузерах (Google, Yandex и т.д.) или в YouTube – это все полнотекстовый поиск.

-Среди механизмов текстового поиска в Postgres можно выделить следующие компоненты:

- Анализаторы
- Словари
- Конфигурации

Разберем каждый из них:

- 1) Анализатор – разделяет документы на фрагменты и классифицирует их:

Хлеб да вода – здоровая еда →

‘вода’: 3, ‘еда’: 5, ‘здоровая’: 4, ‘хлеб’: 1, ‘да’: 2

- 2) Словарь – преобразует фрагменты в лексемы, приводит их к нормализованному виду, удаляет стоп-слова:

Хлеб да вода – здоровая еда →

‘вода’: 3, ‘еда’: 5, ‘здоровая’: 4, ‘хлеб’: 1, ‘да’: 2 →

‘вода’: 3, ‘еда’: 5, ‘здоровая’: 4, ‘хлеб’: 1

- 3) Конфигурация – задает связку анализатора с набором словарей, которые будут обрабатывать выделенные фрагменты (postgresql.conf). Для каждого типа фрагментов в конфигурации присваивается отдельный список словарей. Фрагмент последовательно проходит через все словари, пока в каком-либо из них слово не будет найдено. Если это стоп-слово, или ни один словарь не смог его распознать, то фрагмент не будет учитываться при индексации

То есть текстовый поиск позволяет: пропускать определенные слова (стоп-слова), обрабатывать синонимы и выполнять сложный анализ слов, например, выделять фрагменты не только по пробелам. Все эти функции управляются конфигурациями текстового поиска. Можно как использовать существующие конфигурации, так и писать свои.

Для упрощения создания конфигураций текстового поиска, они строятся из более простых объектов. В PostgreSQL есть четыре типа таких объектов:

- Анализаторы текстового поиска - разделяют документ на фрагменты и классифицируют их (например, как слова или числа)
- Словари текстового поиска – приводят фрагменты к нормализованной форме и отбрасывают стоп-слова
- Шаблоны текстового поиска – предоставляют функции, образующие реализацию словарей (при создании словаря просто задается шаблон и набор параметров для него)

- Конфигурации текстового поиска – выбирают анализатор и набор словарей, который будет использоваться для нормализации фрагментов, выданных анализатором

Анализаторы и шаблоны текстового поиска строятся из низкоуровневых функций на языке C

Основные типы данных:

- `tsvector` – нужен для хранения подготовленных документов (создает лексемы). Для нужд текстового поиска каждый документ должен быть сведен к специальному формату `tsvector`. Поиск и ранжирование выполняются исключительно с этим представлением документа – исходный текст потребуется извлечь, только когда документ будет отобран для вывода пользователю. Поэтому под `tsvector` часто подразумевается документ, тогда как этот тип, конечно, содержит только компактное представление всего документа.
- `tsquery` – для представления обработанных запросов

С этими типами работает целый ряд функций и операторов, и наиболее важный из них – оператор соответствия `@@`.

-Разберем теперь как строятся **индексы**:

Полнотекстовая индексация заключается в предварительной обработке документов и сохранения индекса для последующего быстрого поиска.

То есть если мы хотим ускорить поиск, то в определенных случаях поможет индексация. Она будет включать в себя:

- Разбор документов на фрагменты (подстроки текста документа). Это функцию выполняет анализатор. При этом, можно как использовать встроенный анализатор, так и написать свой узконаправленный
- Преобразование фрагментов в лексемы (слова, имеющие ценность для индексации и поиска). Этот шаг выполняют словари. Можно использовать существующие словари, а также писать свои
- Хранение преобразованных документов, подготовленных для поиска. Например, каждый документ может быть представлен в виде сортированного массива нормализованных лексем. Помимо лексем часто желательно хранить информацию об их положении для ранжирования по близости, чтобы документ, в котором слова расположены “плотнее”, получал более высокий рейтинг, чем документ с разбросанными словами

Лексема – это нормализованный фрагмент, в котором разные словоформы приведены к одной. Например, при нормализации буквы верхнего регистра приводятся к нижнему, а из слов обычно убираются окончания. Благодаря этому можно находить разные формы одного слова, не вводя вручную все возможные варианты. Кроме того, на данном шаге обычно исключаются стоп-слова.

В Postgres для полнотекстового поиска существуют индексы двух видов:

- GIN – рассчитан на случаи, когда чаще выполняется чтение, а не запись. Хранит не записи целиком, а отдельные лексемы со списком места вхождения. Лексемы структурированы в виде b-дерева.
- GIST – наоборот, когда нужно больше записывать в базу данных, чем читать. Данный индекс быстро обновляется, но из-за особенностей его реализации, поиск с таким индексом допускает неточности, которые Postgres исправляет дополнительными проверками. Из-за этого поиск идет дольше. Основан на хэш-таблицах

Рассмотрим, как строится индекс GIN на следующем примере:

Хлеб да вода – здоровая еда.

У кого работа, у того и хлеб.

Слово к ответу, а хлеб – к обеду.

Работа не волк – в лес не убежит.

Преобразованные к типу tsvector – они будут выглядеть так:

0,1 'вод': 3, 'ед': 5, 'здоров': 4, 'хлеб': 1

0,2 'ког': 2, 'работ': 3, 'хлеб': 7

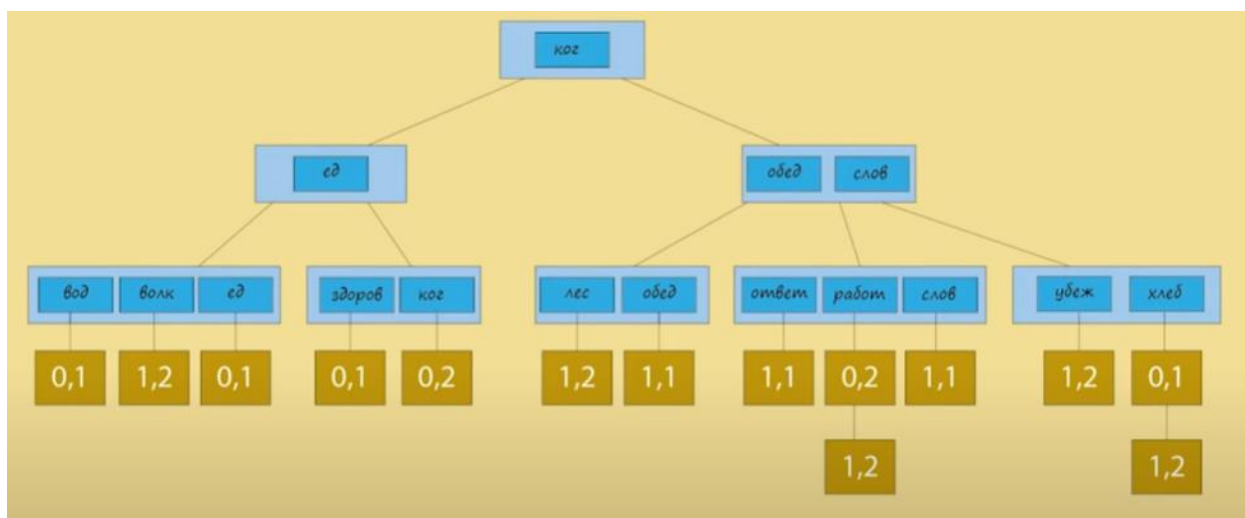
1,1 'обед': 7, 'ответ': 3, 'слов': 1, 'хлеб': 5

1,2 'волк': 3, 'лес': 5, 'работ': 1, 'убеж': 7

Во внутреннем представлении записей базы данных имеются уникальные идентификаторы TID: {[0,1], [0,2], [1,1], [1,2]}

Они состоят из номера блока и номера строки в блоке.

-Внутри Postgres созданный индекс будет иметь вид b-дерева, в узлах которого находятся лексемы, а к листьям крепятся списки TID:



-Полнотекстовый поиск – это не только получения документов, соответствующих запросу, но и возможность оценить, где это соответствие больше, а где меньше – проранжировать результаты.

Ранжирование – это сортировка по степени соответствия запросу, то есть по релевантности. Для ранжирования используется функция `ts_rank`. Для назначения `tsvector` элементам разных весов – используется функция `setweight`. Вес элемента задается буквами A, B, C и D. Обычно это применяется для обозначения важности слов в разных частях документа от самого важного до наименее важного, где A – самый важный, D – самый неважный. Затем эта информация может использоваться при ранжировании результатов поиска.

Преимущества FTS:

- FTS поддерживает лингвистический функционал. Становится возможным работа с различными формами слов и с синонимами
- FTS делает возможным получение результатов поиска на основе их релевантности запросу, что очень эффективно, когда имеется сотни подходящих документов
- FTS позволяет быстрее получать данные за счет возможности использовать индексирования
- FTS позволяет получает данные более оптимизированным способом за счет различных техник, например, удаление стоп-слов, стемминг и т.д.

Недостатки FTS:

- Внедрение и настройка FTS может оказаться довольно сложной, особенно для достижения высокой производительности на больших объемах данных
- FTS требователен к ресурсам системы из-за необходимости обрабатывать и анализировать большие объемы текстовых данных

Поиск текста через SQL:

-Для того, чтобы реализовать текстовый поиск в базе данных – можно просто использовать SQL. Для этого существуют следующие операторы:

- ~
- ~*
- LIKE
- ILIKE

Операторы LIKE и ILIKE используются для поиска подстрок внутри текстовых строк. LIKE чувствителен к регистру, в то время как ILIKE – нет. Они поддерживают символы % (для совпадения с любым числом символов) и _ (для совпадения с одним символом). Эти операторы хорошо подходят для простых запросов, когда нужно найти строки, содержащие определенные фрагменты текста.

Операторы ~ и ~* используются в PostgreSQL для выполнения регулярных выражений. Оператор ~ чувствителен к регистру, а ~* - нет. Данные операторы позволяют сделать в разы больше, чем операторы LIKE и ILIKE, но они требуют глубоких знаний регулярных выражений. Также писать регулярные выражения может быть неэффективно, особенно при создании сложных запросов (очень тяжело предусмотреть и описать все случаи для сложного запроса через одни регулярные выражения), а также при работе с большим текстовым объемом (будет очень страдать производительность).

Также данные операторы не поддерживают лингвистический функционал. Даже регулярные выражения очень ограничены – они не рассчитаны на работу со словоформами – например, “подходят” и “подходить”. С ними возможно пропустить документы, которые содержат “подходят”, но, вероятно, и они представляют интерес при поиске по ключевому слову “подходить”. Конечно, можно попытаться перечислить в регулярном выражении все варианты слова, но это будет очень трудоёмко и подвержено ошибкам, так как некоторые слова могут иметь десятки словоформ.

Преимущества поиска текста через SQL:

- Операторы SQL для поиска текста легко внедрять и использовать
- Данные операторы очень хороши именно для определенного случая – когда поиск текста осуществляется через простые запросы, где не требуется поддержки лингвистического функционала, не требуется работы с большими текстовыми объемами, не требуется упорядочивать результаты поиска по релевантности – здесь данные операторы очень подходят. И в таких случаях – наилучшим решением будет использовать именно данные операторы

Недостатки поиска текста через SQL:

- Такой метод не поддерживает поиск по формам слова. Конечно, можно использовать регулярные выражения, но придется пересмотреть десятки вариантов, что весьма трудоемко, а также легко ошибиться
- Полученные результаты не будут отсортированы по релевантности
- Поиск будет идти медленно, потому что нет полноценной индексной поддержки – при каждом поиске придется просматривать все документы

Отличие FTS от поиска текста через SQL:

-Отличие двух этих методов заключается в их подходах к поиску и обработке текстовых данных. Вот основные различия:

- FTS использует лингвистический анализ для улучшения поиска, в то время, как операторы SQL работают с текстом прямолинейно, без учета морфологии или контекста слова
- FTS позволяет использовать ранжирование результатов по их совпадению запросу, чего не могут операторы SQL
- Операторы SQL – позволяют реализовать поиск по шаблонам с небольшими объемами данных, а также простыми запросами
- FTS – позволяет реализовать сложные запросы с большими объемами текстовых данных
- FTS лучше в плане производительности, за счет его механизмов поиска

Итог:

-Было рассмотрено два способа поиска текста – Postgres FTS и операторы SQL (операторы LIKE, ILIKE, ~, ~*). Выбор метода зависит от требований к приложению. Если в приложении нужен функционал, при котором следует искать простые подстроки без учета морфологии или текста, когда работа идет с маленьким объемом данных и простыми запросами, то в таком случае оптимальным решением будет использование операторов SQL, то есть операторов LIKE, ILIKE, ~, ~*. Однако если же приложение требует более сложного функционала – работа с большими текстовыми полями, нужно учитывать различные формы слова или искать синонимы, нужно ранжировать результаты поиска по их совпадению с запросом, нужна лучшая производительность, то в таком случае SQL просто не предоставляет решения для реализации заданного функционала. В таком случае – нужно реализовывать FTS.

Решение:

-Полнотекстовый поиск будет использоваться только для таблицы постов пользователей – Post. Для всех остальных таблиц: User, UserContact, Subscription – при соответствующем методе поиска текста, будет по-прежнему использован оператор LIKE. Такое решение было принято, так как большие данные есть только в таблице Post – это колонка content, отвечающая за содержимое поста. Также при поиске текста в постах – важно поддерживать поиск различных форм слов, а также выдачу результатов по их степени соответствия запросу. Во всех остальных же таблицах – при соответствующих методах поиска текста, будет происходить работа с маленькими данными, где не требуется ранжировка по релевантности, а также не нужно учитывать морфологию и контекст слова. Здесь нужен простой поиск заданной подстроки в небольшом тексте, с чем прекрасно справится оператор LIKE.

-Для реализации полнотекстового поиска – сначала нужно выбрать колонки из базы данных, по которым будет осуществляться поиск. В нашем случае – полнотекстовый поиск можно будет проводить по title, или content, или сразу по title и content (title и content – колонки таблицы Post).

-Для выполнения полнотекстового поиска, нужно отправить соответствующий SQL-запрос. Данный SQL-запрос всегда должен сопоставлять поисковой вектор с запросом. В таком случае – это будет полнотекстовый поиск. Поисковой вектор – это тип tsvector, который мы ранее разобрали (Напомним, чтобы реализовать полнотекстовый поиск – сначала нужно выбрать колонки из базы данных, по которому будет идти поиск, затем преобразовать эти колонки к типу tsvector, который является массивом лексем с информацией об их позиции, а после сопоставлять этот tsvector с запросом, каждый раз, когда данный запрос приходит). Чтобы создать поисковой вектор tsvector, существует специальная команда to_tsvector. В нашей реализации полнотекстовый поиск осуществляется по колонкам title и content. Чтобы нам каждый раз, когда пользователь захочет выполнить полнотекстовый поиск, не пришлось приводить title и content к типу tsvector – создадим специальную колонку text_tsv в таблице Post. Данная колонка будет

содержать title и content, приведенные к типу tsvector. В таком случае – когда пользователь захочет выполнить полнотекстовый поиск, нам будет достаточно сопоставить запрос с колонкой text_tsv, и не нужно будет выполнять операцию to_tsvector перед сопоставлением.

-Чтобы реализовать данную логику, напомним миграцию, которая все это реализует (./src/database/migrations/20240719151942-add_text_tsv_column_to_post_table.js):

```
src > database > migrations > JS 20240719151942-add_text_tsv_column_to_post_table.js > up
1  const { Sequelize } = require('sequelize')
2
3  async function up({ context: queryInterface }) {
4    const tableInfo = await queryInterface.describeTable('Posts');
5    if (!tableInfo.text_tsv) {
6      await queryInterface.addColumn('Posts', 'text_tsv', {
7        type: Sequelize.TSVECTOR,
8        allowNull: true,
9      });
10 }
```

В функции up данной миграции, мы сначала проверяем – существует ли в таблице Post колонка text_tsv. Если такой колонки не существует, то мы ее создаем для таблицы Post. После создания данной колонки, у всех существующих данных – колонка text_tsv будет равна null. Нужно это исправить:

```
await queryInterface.bulkUpdate('Posts', {
  text_tsv: Sequelize.literal(
    `setweight(to_tsvector('russian', "title"), 'A') || setweight(to_tsvector('russian', "content"), 'B')`
  )
}, {});
```

Как уже было сказано ранее – text_tsv должен содержать поисковой вектор tsvector, основанный на колонках title и content. Здесь мы для всех записей, которые уже существуют в таблице Post – изменяем колонку text_tsv следующим образом:

- to_tsvector('russian', "title") – преобразуем запись title к типу tsvector, указывая, что текст должен быть обработан с учетом особенностей русского языка
- setweight(to_tsvector('russian', "title"), 'A') – устанавливаем вес A для полученных от to_tsvector('russian', "title") лексем. Таким образом, мы устанавливаем метку о том, что все лексемы, связанные с заголовком поста – являются самыми важными (это можно использовать при ранжировании)

- `to_tsvector('russian', "content")` - преобразуем запись `content` к типу `tsvector`, указывая, что текст должен быть обработан с учетом особенностей русского языка
- `setweight(to_tsvector('russian', "content"), 'B')` - устанавливаем вес `B` для полученных от `to_tsvector('russian', "content")` лексем. Таким образом, мы устанавливаем метку о том, что все лексемы, связанные с текстом поста – являются вторыми по важности, после лексем-заголовков (это можно использовать при ранжировании)
- `setweight(to_tsvector('russian', "title"), 'A') || setweight(to_tsvector('russian', "content"), 'B')` – объединяем полученные лексемы-заголовки с полученными лексемами-текстами

Таким образом, мы запишем для каждой записи в таблице `Post`, новое значение колонки `text_tsv` - приведенные к `tsvector` колонки: `title` и `content`.

-Теперь мы сможем использовать `text_tsv` каждый раз, когда пользователь захочет выполнить полнотекстовый поиск. При этом нам не нужно будет выполнять `to_tsvector` перед сопоставлением. Однако есть важный нюанс – пользователь может изменить свой пост. В том числе, он может изменить `title` и `content` поста. В таком случае – нужно будет также изменить и `text_tsv` поста, так как он строится на данных `title` и `content`. Также пользователь может создать новый пост, поэтому нужно будет заполнить `text_tsv` нового поста, поскольку оно будет содержать значение `null` после создания (нет значения по умолчанию).

-Чтобы решить этот вопрос – создадим триггер через данную миграцию, который будет обновлять `text_tsv` посту, если пользователь изменяет `title` или `content` посту, либо же пользователь пытается создать новый пост:

```
await queryInterface.sequelize.query(`
  CREATE OR REPLACE FUNCTION update_text_tsv()
  RETURNS TRIGGER AS $$
  BEGIN
    IF NEW.title IS NOT NULL OR NEW.content IS NOT NULL THEN
      NEW.text_tsv := setweight(to_tsvector('russian', NEW.title), 'A') ||
        setweight(to_tsvector('russian', NEW.content), 'B');
    END IF;
    RETURN NEW;
  END;
  $$ LANGUAGE plpgsql;
`);
```

Здесь мы создаем функцию `update_text_tsv`, которую будет использовать триггер для обновления `text_tsv`. В данной функции есть условие проверки:

`IF NEW.title IS NOT NULL OR NEW.content IS NOT NULL`

эта проверка нужна для случая, когда пользователь обновляет пост. Здесь проверяется – изменяет ли пользователь title или content. Если хоть что-то из этого изменяется, то в таком случае text_tsv обновляется на основе новых значений title и content:

```
NEW.text_tsv := setweight(to_tsvector('russian', NEW.title), 'A') ||  
                  setweight(to_tsvector('russian', NEW.content), 'B')
```

Если же ни title, ни content не обновляются, то в таком случае не обновляется и text_tsv. В случае, когда пользователь будет создавать новый пост – title и content точно будут заданы. Поэтому это условие выполнится и text_tsv заполнится соответствующим образом.

Дальше создадим сам триггер:

```
await queryInterface.sequelize.query(`  
  CREATE TRIGGER update_text_tsv_trigger  
  BEFORE INSERT OR UPDATE ON \"Posts\"  
  FOR EACH ROW EXECUTE PROCEDURE update_text_tsv();  
`);
```

Данный триггер update_text_tsv_trigger – будет выполнять созданную ранее функцию update_text_tsv, если в таблицу Post добавляется новая запись, или изменяется старая. Причем, функция update_text_tsv отработает до создания поста, либо же до изменения поста.

-После создания триггера обновления text_tsv – осталось предусмотреть оптимизацию поиска. Как было описано ранее – для этого существуют индексы. То есть нам нужно создать индекс GIN для колонки text_tsv, чтобы полнотекстовый поиск выполнялся быстрее. Для этого добавим в нашу миграцию также и создание индекса:

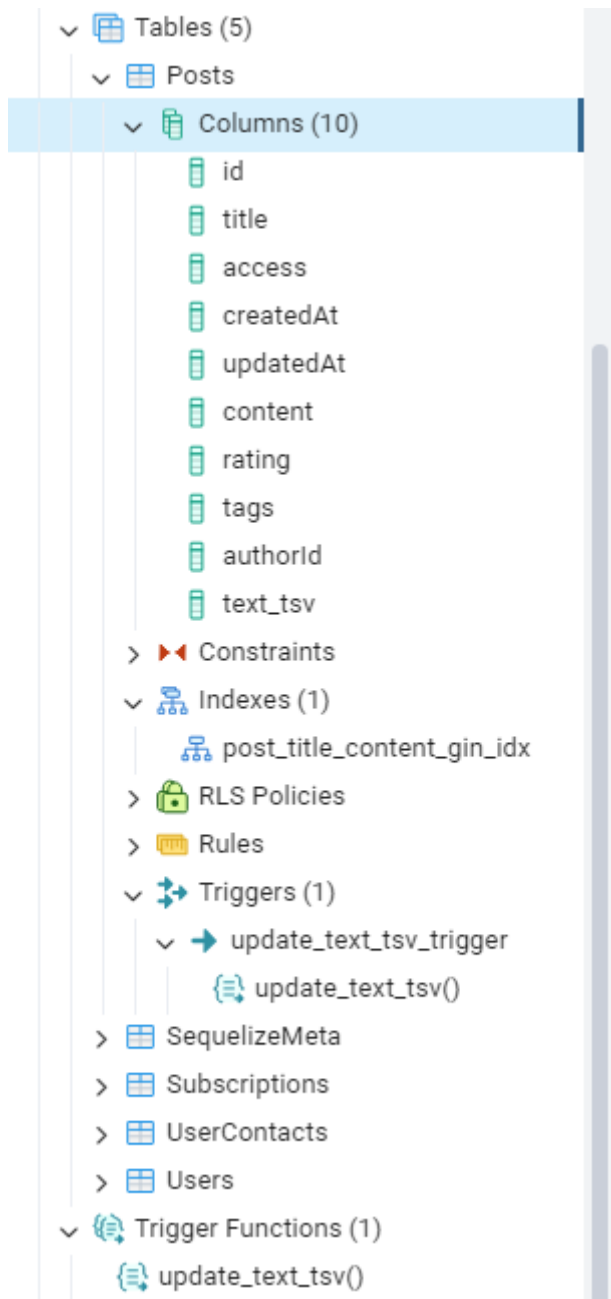
```
await queryInterface.addIndex('Posts',  
  [  
    'text_tsv'  
  ],  
  {  
    using: 'gin',  
    unique: false,  
    name: 'post_title_content_gin_idx'  
  })
```

Здесь мы создали индекс GIN `post_title_content_gin_idx` для колонки `text_tsv`. Я решил использовать именно GIN, а не GIST, так как GIST работает дольше, чем GIN.

-В конце осталось только определить `down` нашей миграции, где мы будем удалять все созданное в `up`: колонку `text_tsv`, функцию `update_text_tsv`, триггер `update_text_tsv_trigger` и индекс `post_title_content_gin_idx`:

```
async function down({ context: queryInterface }) {
  const tableInfo = await queryInterface.describeTable('Posts');
  if (tableInfo.text_tsv) {
    await queryInterface.dropTrigger('Posts', 'update_text_tsv_trigger');
    await queryInterface.removeIndex('Posts', 'post_title_content_gin_idx');
    await queryInterface.removeColumn('Posts', 'text_tsv');
    await queryInterface.dropFunction('update_text_tsv', []);
  }
}
```

-Мы создали миграцию. Теперь применим ее и убедимся, что все было добавлено правильно:



Как можно заметить, все данные были успешно добавлены в таблицу Post: колонка text_tsv, функция update_text_tsv, триггер update_text_tsv_trigger и индекс post_title_content_gin_idx.

-После успешного применения данной миграции – необходимо также добавить text_tsv в модель Post (./src/database/models/post/post.model.ts) для корректной работы Sequelize:

```
@Column({
  type: DataType.TSVECTOR,
  allowNull: true
})
text_tsv?: string | null | undefined;
```

-Сейчас мы добавили в таблицу Post колонку text_tsv, которая содержит tsvector колонок, по которым будет выполняться полнотекстовый поиск. Мы настроили text_tsv на автоматическое обновление через триггер в случае, если изменятся данные колонок, на которых основан text_tsv или создастся новая запись, содержащая text_tsv. Мы добавили индекс GIN для text_tsv для обеспечения быстрого поиска. Теперь нам необходимо принимать запросы от пользователя на полнотекстовый поиск и выполнять их. Для этого нужно изменить контроллер PostController (./src/domain/post/post.controller.ts). Однако перед тем, как переходить к изменению контроллера, также потребуется изменить другие части ПО:

1. Нужно изменить метод GET для /api/posts/public (Get all user posts) в Swagger-документации (./src/middleware/swagger/swagger.yaml):

```
/api/posts/public:
  get:
    summary: Get all user posts
    tags:
      - posts
    security:
      - BearerAuth: []
    parameters:
      - in: query
        name: searchParam
        schema:
          type: string
          enum:
            - title
            - content
          required: false
        description: Search parameter that sets among whom the search substring will be searched - titles or cor
      - in: query
        name: searchString
        schema:
          type: string
```

По заданию – пользователь может решать по какой колонке проводить полнотекстовый поиск:

```
копировать
postsApi?filters[title]=стены
```

Поэтому нужно изменить метод получения всех постов в Swagger-документации так, чтобы он предоставлял возможность производить поиск по определенной колонке. Здесь мы именно это и сделали – добавили в параметры дополнительное поле – searchParam, которое может принимать только два значения: 'title' или

‘content’. Теперь в Swagger-документации – можно отправлять запрос на полнотекстовый поиск постов, выбирая колонку для поиска. Если в searchParam указать ‘title’, то полнотекстовый поиск будет проходить только по колонке title. Если же указать в searchParam ‘content’, то полнотекстовый поиск будет проходить только по колонке content. Если же вообще ничего не указывать в searchParam, то полнотекстовый поиск будет проходить и по ‘title’, и по ‘content’

2. В контроллере нам потребуется написать SQL-запрос, в котором будет происходить сопоставление поискового вектора с запросом. Если не использовать “сырой” SQL, а использовать метод FindAll Sequelize, то в таком случае существует единственный способ написать такой запрос – передать в объект where метода FindAll литерал Sequelize, в котором будет прописан данный SQL-код. Мы воспользуемся как-раз таким способом – использование метода FindAll Sequelize вместо “сырого” SQL-запроса. Для этого нам нужно изменить сервис PostService (./src/domain/post/post.service.ts) таким образом, чтобы он правильно внедрял литерал Sequelize в объект для метода FindAll:

```
private findOptionsJoinLiteral(findOptions: FindOptions, literalQuery: string) {
  findOptions.where = [{
    [Op.and]: [
      {
        ...findOptions.where
      },
      Sequelize.literal(literalQuery)
    ]
  }];

  return findOptions
}
```

```
private setupFindOptions(
  findOptions: FindOptions,
  user: IUserPayload | undefined,
  literalQuery: string | undefined
) {
  if (user) {
    findOptions = this.findOptionsRoleFilter(findOptions, user);
  }

  if (literalQuery) {
    findOptions = this.findOptionsJoinLiteral(findOptions, literalQuery);
  }

  return findOptions;
}
```

```

public async getAllUserPosts(
  findOptions: FindOptions,
  user?: IUserPayload,
  literalQuery?: string
): Promise<Post[]>
{
  findOptions = this.setupFindOptions(findOptions, user, literalQuery);
  const posts = await Post.findAll(findOptions);
  return posts;
}

```

Была добавлена новая функция `findOptionsJoinLiteral`, которая безопасно внедряет написанный литерал Sequelize в объект `findOptions` для метода `findAll` Sequelize. Также была добавлена новая функция `setupFindOptions`, которая будет настраивать соответствующим образом объект `findOptions` (добавлять туда данные на основе ролей пользователя если нужно, встраивать туда литерал Sequelize если нужно). Данный метод просто выносит логику настройки объекта `findOptions` для методов Sequelize, чтобы не пришлось в методах сервиса повторно писать настройку `findOptions`. Также был изменен сам метод `getAllUserPosts`. Этот метод будет использоваться, когда пользователь будет проводить полнотекстовый поиск по постам. Данный метод теперь принимает необязательный параметр `literalQuery`, который будет представлять из себя написанный литерал Sequelize. Дальше метод настроит объект `findOptions` через `setupFindOptions`, а после настройки – выполнит метод Sequelize `FindAll` на основе данного `findOptions` и вернет массив постов.

3. В предыдущем пункте не было реализовано все, что необходимо для работы полнотекстового поиска со стороны сервиса `PostService`. Нам также нужно предусмотреть ситуации, когда пользователь захочет искать не по двум столбцам одновременно – `title` и `content`, а по одному из столбцов – либо по `title`, либо по `content`. Также нужно предусмотреть ситуацию, когда пользователь будет вводить в поиск не одно слово, а несколько слов. Для решения данных задач, внедрим в сервис `PostService` следующее:

- `ftsParamWeightSettings`:

```

private readonly ftsParamWeightSettings = {
  title: 'A',
  content: 'B'
};

```

- setupFTSParams:

```

78 public setupFTSParams(searchParam: string, searchString: string): string {
79     if (searchParam === 'title' || searchParam === 'content') {
80         const searchWeight = this.ftsParamWeightSettings[searchParam];
81         return searchString.split(/\s+/).map(word => `${word}:${searchWeight}`).join(" & ");
82     }
83
84     return searchString.split(/\s+/).join(' & ');
85 }

```

Поле ftsParamWeightSettings будет служить конфигуратором для фильтрации поиска по определенной колонке. Функция setupFTSParams – отвечает за фильтрацию по определенной колонке, а также за возможность совершить поиск по нескольким словам. Данная функция принимает два аргумента:

- searchParam – столбец, где пользователь хочет проводить поиск (либо ‘title’, либо ‘content’)
- searchString – то, что хочет найти пользователь (любая строка)

В функции происходит проверка - равен ли searchParam значению ‘title’ или значению ‘content’. Если равен, то в таком случае пользователь не хочет проводить поиск одновременно по столбцам title и content, а хочет провести поиск по одному из этих столбцов - если в searchParam был установлен ‘title’, то пользователь хочет выполнить полнотекстовый поиск только по колонке ‘title’. Если же в searchParam был установлен ‘content’, то пользователь хочет выполнить полнотекстовый поиск только по колонке ‘content’.

Напомню, что в векторе поиска text_tsv записаны лексемы как для title, так и для content. Соответственно, если мы хотим выполнить поиск только по title, то нужно получить из text_tsv только лексемы title, и наоборот для content. В этом нам помогут веса лексем – одна из причин, по которой мы устанавливали в миграции веса A, B для лексем через setweight. Мы воспользуемся метками полнотекстового поиска. Например, если пользователь ищет по слову “работа” и хочет искать именно в лексемах с весом A, то нам достаточно просто передать в запрос следующее:

“работа:A”

Таким образом, запрос будет искать именно по лексемам с весом A. Аналогично будет для весов B, C и D. Напомню, что мы установили для лексем title вес A, а для лексем content вес B. Именно данную логику мы здесь и реализуем:

```

if (searchParam === 'title' || searchParam === 'content') {
    const searchWeight = this.ftsParamWeightSettings[searchParam];
    return searchString.split(/\s+/).map(word => `${word}:${searchWeight}`).join(" & ");
}

```

В `searchWeight` добавляется нужная метка за счет использования конфигурации `ftsParamWeightSettings` (именно поэтому мы создали такую конфигурацию, и именно поэтому мы создали ее именно такой).

Стоит отметить, что добавлять метки в поиске — это накладно в плане производительности. Однако производительность не должна пострадать, так как мы добавили для `text_tsv` индекс GIN.

После получения метки, по которой хочет искать пользователь — строка поиска пользователя разбивается по пробелам:

```
searchString.split(/\s+/)
```

Затем с помощью `map` к каждому слову добавляется данная метка. А уже после с помощью `join` — разобранная строка объединяется в одну через `&`. Как итог — метод вернет преобразованную строку.

Если же пользователь захотел искать именно по колонкам `title` и `content` одновременно, то в таком случае пользователь не передаст `searchParam`. То есть данная проверка не пройдет:

```
if (searchParam === 'title' || searchParam === 'content') {  
  const searchWeight = this.ftsParamWeightSettings[searchParam];  
  return searchString.split(/\s+/).map(word => `${word}:${searchWeight}`).join(" & ");  
}
```

В таком случае нам не нужно к каждому слову добавлять метку поиска, поэтому в таком случае метод выполнит такие же действия, но без добавления меток:

```
return searchString.split(/\s+/).join(' & ');
```

То есть — если пользователь захотел искать по следующей строке:
“В Солнечной системе всего 8 планет”

То в таком случае данный метод преобразует данную строку следующим образом:
“В & Солнечной & системе & всего & 8 & планет”

Если пользователь захотел искать по этой же строке, но при этом только в заголовках поста, то данный метод преобразует данную строку следующим образом:

“В:A & Солнечной:A & системе:A & всего:A & 8:A & планет:A”

Если пользователь захотел искать по этой же строке, но при этом только в текстах поста, то данный метод преобразует данную строку следующим образом:

“В:B & Солнечной:B & системе:B & всего:B & 8:B & планет:B”

Таким образом — мы будем получать только определенные посты, полученные через лексемы с определенным весом.

Также важно объяснить – зачем мы вставили & между словами. Все дело в том, что, когда будет выполняться полнотекстовый поиск – будет выполняться определенный SQL-запрос. Чтобы этот SQL-запрос выполнял именно полнотекстовый поиск – нам потребуется использовать функцию `to_tsquery` в запросе. Данная функция будет принимать в качестве аргумента поисковой запрос пользователя. Данная функция устроена так, что она не может работать со строкой, которая содержит несколько слов. В таком случае будет ошибка синтаксиса. Чтобы избежать этой ошибки – нужно соединять слова операторами. В данном случае, мы используем оператор &. Это буквально означает, что мы проводим поиск, по нескольким словам, одновременно. То есть:

“Кирпичная & стена & дома” – это будет означать: найти все посты, в которых одновременно будут слова ‘кирпичн’, ‘стен’, ‘дом’ (функция `to_tsquery` также преобразует данную строку к лексемам)

“Кирпичная:A & стена:A & дома:A” – это будет означать: найти все посты, в чьих заголовках одновременно будут слова ‘кирпичн’, ‘стен’, ‘дом’ (функция `to_tsquery` также преобразует данную строку к лексемам)

“Кирпичная:B & стена:B & дома:B” – это будет означать: найти все посты, в чьих текстах одновременно будут слова ‘кирпичн’, ‘стен’, ‘дом’ (функция `to_tsquery` также преобразует данную строку к лексемам)

Подробнее функцию `to_tsquery`, а также запрос полнотекстового поиска мы разберем в следующем пункте

4. Теперь можно изменять сам `PostController`. Изменим метод `getAllUserPosts`, который возвращает все посты пользователя. Именно данный метод позволяет выполнить полнотекстовый поиск:

```

public async getAllUserPosts(req: Request, res: Response, next: NextFunction): Promise<Response | void> {
  try {
    const user = req.user as IUserPayload;
    const searchParam = req.query.searchParam as string;

    let searchString = req.query.searchString as string;
    let posts = [];

    if (!searchString) {
      posts = await this.postService.getAllUserPosts({}, user);
    }
    else {
      searchString = this.postService.setupFTSParams(searchParam, searchString);
      posts = await this.postService.getAllUserPosts(
        {
          order: [
            [
              Sequelize.literal(`ts_rank(text_tsv, to_tsquery('russian', '${searchString}'))`),
              'DESC'
            ]
          ],
          user,
          `text_tsv @@ to_tsquery('russian', '${searchString}'))`
        },
      );
    }

    return res.status(200).json({ status: 200, data: posts, message: "List of all posts" });
  }
  catch (err) {
    next(err);
  }
}

```

Здесь:

- Получаем пользователя user, который выполняет метод, получаем параметр запроса searchParam, который задает колонку поиска, получаем строку запроса searchString для полнотекстового поиска, слова которой будем искать в базе данных, а также определяем массив постов, который вернем пользователю после поиска:

```

const user = req.user as IUserPayload;
const searchParam = req.query.searchParam;

let searchSubstring = req.query.searchSubstring;
let posts = [];

```

- Далее мы проверяем – задана или нет строка для поиска – if (!searchString). Если она не задана, то это означает, что пользователь не применяет полнотекстовый поиск, а просто хочет получить список всех своих постов. В таком случае – выполняется соответствующий запрос:

```

posts = await this.postService.getAllUserPosts({}, user);

```

После выполнения данного запроса – пользователю вернется полученный результат.

Если же строка `searchSubstring` для поиска задана, то это означает, что пользователь хочет выполнить полнотекстовый поиск по данной строке. В таком случае – создается специальный запрос:

```
searchString = this.postService.setupFTSParams(searchParam, searchString);
posts = await this.postService.getAllUserPosts(
  {
    order: [
      [
        Sequelize.literal(`ts_rank(text_tsv, to_tsquery('russian', '${searchString}'))`),
        'DESC'
      ]
    ],
    user,
    `text_tsv @@ to_tsquery('russian', '${searchString}')`
  );
```

Здесь сначала проверяется – как пользователь хочет выполнить полнотекстовый поиск за счет созданного ранее метода `setupFTSParams` сервиса `PostService`:

```
searchString = this.postService.setupFTSParams(searchParam, searchString);
```

Как было описано ранее – данный метод приведет строку поиска к нужному формату – при необходимости добавит веса, при необходимости разделит все слова с помощью `&`.

Дальше создается сам запрос:

```
posts = await this.postService.getAllUserPosts(
  {
    order: [
      [
        Sequelize.literal(`ts_rank(text_tsv, to_tsquery('russian', '${searchString}'))`),
        'DESC'
      ]
    ],
    user,
    `text_tsv @@ to_tsquery('russian', '${searchString}')`
  );
```

Здесь создается литерал `Sequelize`:

```
`text_tsv @@ to_tsquery('russian', '${searchString}')`
```

Именно данный код отвечает за полнотекстовый поиск в SQL-запросе. Здесь мы берем колонку `text_tsv` типа `tsvector`, которая содержит лексемы для поиска. Также мы преобразуем строку поиска `searchString` к `tsquery`:

```
to_tsquery('russian', '${searchString}')
```

Это специальный тип, к которому нужно привести строку запроса, чтобы ее можно было сопоставить с вектором поиска (создается такой же массив лексем, как в векторе поиска, чтобы можно было сопоставить запрос с вектором поиска).

Дальше просто происходит сопоставление вектора поиска `text_tsv` с запросом за счет оператора сопоставления `@@`. За счет этого – получается находить нужные записи из базы данных, которые совпадают с запросом, через полнотекстовый поиск.

Сервис дальше встроит этот литерал `Sequelize` в `WHERE` запроса.

Также мы указывает `order`:

```
order: [
  [
    Sequelize.literal(`ts_rank(text_tsv, to_tsquery('russian', '${searchSubstring}'))`),
    'DESC'
  ]
]
```

Он нужен, чтобы проранжировать по релевантности результаты. Для этого используется `ts_rank`. Эта функция принимает вектор сопоставления и запрос. Она высчитывает коэффициент релевантности для каждого результата. Соответственно поэтому мы используем литерал `Sequelize` для вычисления `ts_rank` каждому результату в совокупности с `ORDER BY DESC`. То есть для каждого результата вычислится коэффициент релевантности, а затем с помощью `ORDER DESC` – результаты отсортируются по убыванию (то есть от результата с наибольшим коэффициентом релевантности до результата с наименьшим коэффициентом). Сервис вставит это в запрос следующим образом:

```
ORDER BY ts_rank(text_tsv, to_tsquery('russian', '${searchSubstring}')) DESC
```

Также еще здесь мы передаем в сервис объект пользователя `user`, чтобы выдавать данные пользователю на основе его роли:


```
posts = await this.postService.getAllUserPosts(
{
  order: [
    [
      Sequelize.literal(`ts_rank(text_tsv, to_tsquery('russian', '${searchSubstring}'))`),
      'DESC'
    ]
  ],
  user,
  `text_tsv @@ to_tsquery('russian', '${searchSubstring}'))`
});
```

Таким образом мы создаем SQL-запрос, который выполнит полнотекстовый поиск на основе запроса пользователя и вернет результат. Данный код составит примерно такой SQL-запрос:

```
SELECT "id", "title", "access", "content", "rating", "tags"
FROM "Posts" AS "Post"
WHERE ("Post"."authorId" = '75db0cf8-32ba-4f04-8095-7dca55052ea0' AND
text_tsv @@ to_tsquery('russian', 'Кирпичная:А & стена:А'))
ORDER BY ts_rank(text_tsv, to_tsquery('russian', 'Кирпичная:В & стена:В'))
DESC;
```

Тестирование:

-Проведем тестирование внедренного функционала полнотекстового поиска постов. Для тестирования воспользуемся OpenApi-спецификацией Swagger. Запустим приложение и перейдем к документации Swagger (<http://localhost:5000/api>).

-Сначала авторизируемся с помощью метода POST /auth/signin:

The screenshot shows the Swagger UI for the 'auth' API. The selected endpoint is 'POST /auth/signin' with the description 'Login to account'. There are no parameters. The request body is set to 'application/json'. An example JSON payload is provided:

```
{
  "email": "awesom-e4000@mail.ru",
  "password": "1qaz!@WSXx"
}
```

. The 'Execute' button is visible at the bottom.

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/auth/signin' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "email": "awesom-e4000@mail.ru",
    "password": "jdHGDjg74hd?!!!"
  }'
```

Request URL

http://localhost:5000/auth/signin

Server response

Code	Details
200	<p>Response body</p> <pre>{ "status": 200, "data": { "name": "jason_statham", "email": "awesom-e4000@mail.ru" }, "message": "Authorization was successful" }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true access-control-allow-origin: * connection: keep-alive content-length: 118 content-type: application/json; charset=utf-8 date: Mon, 22 Jul 2024 03:33:24 GMT etag: W/"76-kFcoMafW41/nDuSn4D1uf31D9So" keep-alive: timeout=5 x-powered-by: Express</pre>

Мы успешно авторизовались.

-Теперь создадим новый пост пользователю с помощью метода POST /api/posts/public. Здесь мы проверим работу триггера – после создания поста, у него должен обновиться соответствующим образом text_tsv:

posts

GET /api/posts/public Get all user posts

POST /api/posts/public Create a new post

Parameters

No parameters

Request body required

application/json

Examples:

[Modified value]

```
{
  "title": "Новый Гарри Поттер",
  "access": "public",
  "content": "Сегодня я хочу обсудить один из самых обсуждаемых на сегодня слухов в индустрии кино. Это просто супер! Я говорю правду!",
  "tags": [
    "фильмы",
    "Гарри Поттер"
  ]
}
```

Execute

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/api/posts/public' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "Новый Гарри Поттер",
    "access": "public",
    "content": "Сегодня я хочу обсудить один из самых обсуждаемых на сегодня слухов в индустрии кино. Это просто супер! Я говорю правду!",
    "tags": [
      "фильмы",
      "Гарри Поттер"
    ]
  }'
```

Request URL

http://localhost:5000/api/posts/public

Server response

Code	Details
201	<div><div>Response body</div><pre>{ "status": 201, "data": { "id": "f684e441-e88c-4cb8-8aff-5b452bd6dbe5", "title": "Новый Гарри Поттер", "access": "public", "content": "Сегодня я хочу обсудить один из самых обсуждаемых на сегодня слухов в индустрии кино. Это просто супер! Я говорю правду!", "rating": 0, "tags": ["фильмы", "Гарри Поттер"] }, "message": "Post successfully created" }</pre><div><div></div>Download</div></div> <div><div>Response headers</div><pre>access-control-allow-credentials: true access-control-allow-origin: * connection: keep-alive content-length: 460 content-type: application/json; charset=utf-8 date: Mon, 22 Jul 2024 03:37:02 GMT etag: W/"1cc-UV1x08rDCNfy6eyFk8MsQvvaYac" keep-alive: timeout=5 location: /api/posts/f684e441-e88c-4cb8-8aff-5b452bd6dbe5 x-powered-by: Express</pre></div>

Как можно заметить – пост успешно создан. Теперь проверим – обновился ли в базе данных `text_tsv` для этого поста:

```
'garr':2A 'говор':22B 'индустр':16B 'кин':17B 'нов':1A 'обсуд':7B 'обсужда':11B 'поттер':3A 'правд':23B 'прост':19B 'сам':10B 'сегодн':4B,13B 'слух':14B 'супер':20B 'хоч':6B 'з...
```

`text_tsv` успешно обновился для новой записи. Это означает, что триггер при создании новой записи в таблице `Post` отработал правильно.

Создадим еще один пост:

posts

GET /api/posts/public Get all user posts

POST /api/posts/public Create a new post

Parameters

No parameters

Request body required

application/json

Examples:

[Modified value]

```
{
  "title": "Хэвон - супер",
  "access": "public",
  "content": "Я влюбился в одну прекрасную девочку..."
}
```

Execute

Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/api/posts/public' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "Хэон - супер",
    "access": "public",
    "content": "Я влюбился в одну прекрасную девочку..."
  }'
```

Request URL

http://localhost:5000/api/posts/public

Server response

Code Details

201

Response body

```
{
  "status": 201,
  "data": {
    "id": "f01bb660-4145-4aef-ac66-c6b517daff35",
    "title": "Хэон - супер",
    "access": "public",
    "content": "Я влюбился в одну прекрасную девочку...",
    "rating": 0,
    "tags": null
  },
  "message": "Post successfully created"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 263
content-type: application/json; charset=utf-8
date: Mon, 22 Jul 2024 03:43:00 GMT
etag: W/"107-AhpA83ivLvgQnMQKwfrOMfoqo"
keep-alive: timeout=5
location: /api/posts/f01bb660-4145-4aef-ac66-c6b517daff35
x-powered-by: Express
```

'влюб':4B 'девочк':8B 'одн':6B 'прекрасн':7B 'супер':2A 'хэон':1A

Еще один пост также создался успешно и text_tsv также обновился должным образом в базе данных. Значит триггер при создании новой записи в таблице Post – работает правильно.

-Теперь попробуем выполнить сам полнотекстовый поиск с помощью метода GET /api/posts/public. Для начала получим список всех постов пользователя (без полнотекстового поиска):

posts

GET /api/posts/public Get all user posts

Parameters

Cancel

Name	Description
searchParam string (query)	Search parameter that sets among whom the search substring will be searched - titles or contents --
searchSubstring string (query)	Search substring that is searched among titles or contents searchSubstring

Execute

Для этого просто не будем ничего вводить в searchParam и searchSubstring:

Responses

Curl

curl -X 'GET' \n'http://localhost:5000/api/posts/public' \n-H 'accept: application/json'

Request URL

http://localhost:5000/api/posts/public

Server response

CodeDetails

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "f01bb660-4145-4aef-ac66-c6b517daff35",
      "title": "Хэмон - супер",
      "access": "public",
      "content": "Я влюбился в одну прекрасную девочку...",
      "rating": 0,
      "tags": null
    },
    {
      "id": "43a6c97b-7a97-41a2-8076-77b97bce4d9f",
      "title": "NMIXX - очень перспективная группа",
      "access": "public",
      "content": "Я влюбился в девочек из NMIXX. Слушая их музыку, мне хочется обнять каждую...",
      "rating": 0,
      "tags": [
        "музыка",
        "круп",
        "nmiXX"
      ]
    },
    {
      "id": "f684a441-e88c-4cb8-8aff-5b452bd6be5",
      "title": "Новый Гарри Поттер",
      "access": "public",
      "content": "Сегодня я хочу обсудить один из самых обсуждаемых на сегодня слухов в индустрии кино. Это просто супер! Я говорю правду!",
      "rating": 0,
      "tags": [
        "фильмы",
        "Гарри Поттер"
      ]
    }
  ],
  "message": "List of all posts"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 986
content-type: application/json; charset=utf-8
date: Mon, 22 Jul 2024 03:47:28 GMT
etag: W/"3da-P6LEwV+H801wvu/Cv1zBmIP21E"
keep-alive: timeout=5
x-powered-by: Express
```

Responses

Curl

curl -X 'GET' \n'http://localhost:5000/api/posts/public' \n-H 'accept: application/json'

Request URL

http://localhost:5000/api/posts/public

Server response

CodeDetails

200

Response body

```
{
  "tags": null
},
{
  "id": "43a6c97b-7a97-41a2-8076-77b97bce4d9f",
  "title": "NMIXX - очень перспективная группа",
  "access": "public",
  "content": "Я влюбился в девочек из NMIXX. Слушая их музыку, мне хочется обнять каждую...",
  "rating": 0,
  "tags": [
    "музыка",
    "круп",
    "nmiXX"
  ]
},
{
  "id": "f684a441-e88c-4cb8-8aff-5b452bd6be5",
  "title": "Новый Гарри Поттер",
  "access": "public",
  "content": "Сегодня я хочу обсудить один из самых обсуждаемых на сегодня слухов в индустрии кино. Это просто супер! Я говорю правду!",
  "rating": 0,
  "tags": [
    "фильмы",
    "Гарри Поттер"
  ]
}
},
"message": "List of all posts"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 986
content-type: application/json; charset=utf-8
date: Mon, 22 Jul 2024 03:47:28 GMT
etag: W/"3da-P6LEwV+H801wvu/Cv1zBmIP21E"
keep-alive: timeout=5
x-powered-by: Express
```

Мы получили все посты пользователя. Значит все работает правильно.

-Теперь выполним полнотекстовый поиск:

posts

GET /api/posts/public Get all user posts

Parameters

Name	Description
searchParam	Search parameter that sets among whom the search substring will be searched - titles or contents
string (query)	--
searchSubstring	Search substring that is searched among titles or contents
string (query)	супер

Execute Clear

Для выполнения полнотекстового поиска просто введем в searchSubstring значение. Также не будем ничего вводить в searchParam. Если задан searchSubstring, а searchParam не задан, то поиск будет проходить по title, и по content:

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/posts/public?searchSubstring=ND1X81ND1X83ND0N8FND0N85ND1X80' \
  -H 'accept: application/json'
```

Request URL

http://localhost:5000/api/posts/public?searchSubstring=ND1X81ND1X83ND0N8FND0N85ND1X80

Server response

Code Details

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "f01bb660-4145-4aef-ac66-c6b517daff35",
      "title": "Хэсон - супер",
      "access": "public",
      "content": "Я влюбился в одну прекрасную девочку...",
      "rating": 0,
      "tags": null
    },
    {
      "id": "f684a441-e88c-4cb8-8aff-5b452bd5dbe5",
      "title": "Норман Гарри Поттер",
      "access": "public",
      "content": "Сегодня я хочу обсудить один из самых обсуждаемых на сегодня слухов в индустрии кино. Это просто супер! Я говорю правду!",
      "rating": 0,
      "tags": [
        "фэнтези",
        "Гарри Поттер"
      ]
    }
  ],
  "message": "List of all posts"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 658
content-type: application/json; charset=utf-8
date: Mon, 22 Jul 2024 03:53:59 GMT
etag: W/"292"/jsOed4UthozrjVcjps5gucx10bE"
keep-alive: timeout=5
x-powered-by: Express
```

Метод отработал верно — мы получили два поста: в одном подстрока “супер” находится в title, а в другом подстрока “супер” находится в content.

Теперь выполним этот же метод, но теперь укажем “title” в searchParam. В таком случае — метод должен вернуть только один пост, где подстрока “супер” находится в title:

posts

GET /api/posts/public Get all user posts

Parameters

Name	Description
searchParam string (query)	Search parameter that sets among whom the search substring will be searched - titles or contents <input type="text" value="title"/>
searchSubstring string (query)	Search substring that is searched among titles or contents <input type="text" value="супер"/>

Execute **Clear**

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/posts/public?searchParam=title&searchSubstring=ND1X81ND1X83ND0NBFD0NB5ND1X80' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public?searchParam=title&searchSubstring=ND1X81ND1X83ND0NBFD0NB5ND1X80
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "status": 200, "data": [{ "id": "f01bb660-4145-4aef-ac66-c6b517daff35", "title": "Хэмон - супер", "access": "public", "content": "Я влюбился в одну прекрасную девочку...", "rating": 0, "tags": null }], "message": "List of all posts" }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true access-control-allow-origin: * connection: keep-alive content-length: 257 content-type: application/json; charset=utf-8 date: Mon, 22 Jul 2024 03:57:29 GMT etag: W/"101-aWajhZgdkgUit4UKeacAdnGncA" keep-alive: timeout=5 x-powered-by: Express</pre>

Метод отработал правильно – вернул только один пост, где подстрока “супер” находится в title.

Теперь выполним этот же метод, но теперь укажем “content” в searchParam. В таком случае – метод должен вернуть только один пост, где подстрока “супер” находится в content:

posts

GET /api/posts/public Get all user posts

Parameters

Name	Description
searchParam string (query)	Search parameter that sets among whom the search substring will be searched - titles or contents <input type="text" value="content"/>
searchSubstring string (query)	Search substring that is searched among titles or contents <input type="text" value="супер"/>

Execute **Clear**

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/posts/public?searchParam=content&searchSubstring=ND1X81ND1X83ND0X8FND0X85ND1X80' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public?searchParam=content&searchSubstring=ND1X81ND1X83ND0X8FND0X85ND1X80
```

Server response

Code	Details
200	<p>Response body</p> <pre>{ "status": 200, "data": [{ "id": "f684a441-e88c-4cb8-8aff-5b452bd6dbe5", "title": "Новый Гарри Поттер", "access": "public", "content": "Сегодня я хочу обсудить один из самых обсуждаемых на сегодня слухов в индустрии кино. Это просто супер! Я говорю правду!", "rating": 0, "tags": ["фильм", "Гарри Поттер"] }], "message": "List of all posts" }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true access-control-allow-origin: * connection: keep-alive content-length: 454 content-type: application/json; charset=utf-8 date: Mon, 22 Jul 2024 03:59:25 GMT etag: W/"1c6-6f43-jCxdGnhqncqzXdyfPlq00" keep-alive: timeout=5 x-powered-by: Express</pre>

Метод отработал правильно – вернул только один пост, где подстрока “супер” находится в content, причем данный пост отличается от поста, полученного в предыдущем эксперименте, так как там “супер” искался в title, а здесь “супер” искался в content.

-Убедимся, что триггер работает правильно и при обновлении поста. Обновим данный пост:

```
"id": "f01bb660-4145-4aef-ac66-c6b517daff35",
"title": "Хэвон - супер",
"access": "public",
"content": "Я влюбился в одну прекрасную девочку...",
"rating": 0,
"tags": null
```

Вот text_tsv данного поста до обновления:

```
'влюб':4В 'девочк':8В 'одн':6В 'прекрасн':7В 'супер':2А 'хэвон':1А
```

Обновим пост с помощью метода PATCH /api/posts/public/{postId}:

posts

GET /api/posts/public Get all user posts

POST /api/posts/public Create a new post

GET /api/posts/public/{postId} Get a post by id

PATCH /api/posts/public/{postId} Update a post by id

Parameters

Cancel

Reset

Name

Description

postId * required

Post Id. Example: 550e8400-e29b-41d4-a716-446655440000

string

(path)

f01bb660-4145-4aef-ac66-c6b517daff35

Request body required

application/json

Examples:

[Modified value]

```
{
  "content": "Хэвон - супер! Я долго наблюдал..."
}
```

Execute

Responses

Curl

```
curl -X 'PATCH' \
'http://localhost:5000/api/posts/public/f01bb660-4145-4aef-ac66-c6b517daff35' \
-H 'accept: application/json' \
-H 'Content-type: application/json' \
-d '{
  "content": "Хэвон - супер! Я долго наблюдал..."
}'
```

Request URL

http://localhost:5000/api/posts/public/f01bb660-4145-4aef-ac66-c6b517daff35

Server response

Code

Details

200

Response body

```
{
  "status": 200,
  "data": {
    "id": "f01bb660-4145-4aef-ac66-c6b517daff35",
    "title": "Хэвон - супер",
    "access": "public",
    "content": "Хэвон - супер! Я долго наблюдал...",
    "rating": 0,
    "tags": null,
    "updatedAt": "2024-07-22T04:06:14.942Z"
  },
  "message": "Post successfully updated"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 290
content-type: application/json; charset=utf-8
date: Mon, 22 Jul 2024 04:06:14 GMT
etag: W/"122-lb/g/N73/Ru032H/1Ae0OfixuA"
keep-alive: timeout=5
x-powered-by: Express
```

Download

Пост успешно создан. Теперь проверим в базе данных, что text_tsv обновился:

```
'долг':6В 'наблюда':7В 'супер':2А,4В 'хэвон':1А,3В
```

Как можно заметить – text_tsv в базе данных действительно обновился. Попробуем получить этот пост через полнотекстовый поиск GET /api/posts/public по новому ключевому слову, который мы добавили, обновив пост:

posts

GET

/api/posts/public

Get all user posts

Parameters

Cancel

Name	Description
searchParam string (query)	Search parameter that sets among whom the search substring will be searched - titles or contents --
searchSubString string (query)	Search substring that is searched among titles or contents долг

ExecuteClear

Responses

Curl

curl -X 'GET' \ 'http://localhost:5000/api/posts/public?searchSubString=ND09KB4ND09KBEND09KB3' \ -H 'accept: application/json'

Request URL

http://localhost:5000/api/posts/public?searchSubString=ND09KB4ND09KBEND09KB3

Server response

Code

Details

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "f01bb660-4145-4aef-ac66-c6b51daff35",
      "title": "Хэмон - супер",
      "access": "public",
      "content": "Хэмон - супер! Я долго наблюдал...",
      "rating": 0,
      "tags": null
    }
  ],
  "message": "List of all posts"
}
```

Download

Response headers

access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 245
content-type: application/json; charset=utf-8
date: Mon, 22 Jul 2024 04:18:24 GMT
etag: W/"45-MEMNIVKE87YJquEauvzE8boFIY"
keep-alive: timeout=5
x-powered-by: Express

Мы получили обновленный пост через новое ключевое слово. Значит триггер при обновлении поста также работает правильно.

-Теперь убедимся, что пользователь может получить только свои посты через полнотекстовый поиск. Для этого сейчас получим случайный пост пользователя через полнотекстовый поиск GET /api/posts/public:

posts

GET

/api/posts/public

Get all user posts

Parameters

Cancel

Name	Description
searchParam string (query)	Search parameter that sets among whom the search substring will be searched - titles or contents --
searchSubString string (query)	Search substring that is searched among titles or contents влюб

ExecuteClear

Responses

Curl

```
curl -X 'GET' \  
  'http://localhost:5000/api/posts/public?searchSubstring=ND00B2ND00B8ND1S8E ND00B1' \  
  -H 'accept: application/json'
```

Request URL

http://localhost:5000/api/posts/public?searchSubstring=ND00B2ND00B8ND1S8E ND00B1

Server response

Code

Details

200

Response body

```
{  
  "status": 200,  
  "data": [  
    {  
      "id": "43a6c97b-7a97-41a2-8076-77b97bce4d9f",  
      "title": "NMIXX - очень перспективная группа",  
      "access": "public",  
      "content": "Я влюбился в девочек из NMIXX. Слушая их музыку, мне хочется обнять каждую...",  
      "rating": 0,  
      "tags": [  
        "музыка",  
        "kpop",  
        "nmixx"  
      ]  
    }  
  ],  
  "message": "List of all posts"
```

Download

Response headers

```
access-control-allow-credentials: true  
access-control-allow-origin: *  
connection: keep-alive  
content-length: 381  
content-type: application/json; charset=utf-8  
date: Mon, 22 Jul 2024 04:13:31 GMT  
etag: W/"17d-kyu4CuFS1F++vX/h+N21VW0EsZ4"  
keep-alive: timeout=5  
x-powered-by: Express
```

Теперь выйдем из текущего аккаунта, зайдем в другой аккаунт, создадим там пост, где будет такое же ключевое слово, по которому мы нашли пост в текущем аккаунте, а после выполним полнотекстовый поиск по этому же ключевому слову. Мы должны получить только новый созданный пост в новом аккаунте. Мы не должны получать пост из текущего аккаунта, так как каждый пользователь может получить только свои посты.

Выйдем из текущего аккаунта с помощью метода GET /auth/logout:

auth

GET /auth/payload Get a user payload

POST /auth/signin Login to account

POST /auth/signup Register a user

GET /auth/logout Logout of account

Parameters

No parameters

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/auth/logout' \
  -H 'accept: */*'
```

Request URL

http://localhost:5000/auth/logout

Server response

Code	Details
200	<p>Response body</p> <pre>{ "status": 200, "message": "Logged out successfully" }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true access-control-allow-origin: * content-length: 50 content-type: application/json; charset=utf-8 date: Mon, 22 Jul 2024 04:17:18 GMT etag: W/"32-uPMLvQo300uq3Id+XL8g9CI6Q" x-powered-by: Express</pre>

Мы успешно вышли из текущего аккаунта. Теперь войдем в новый:

auth

GET /auth/payload Get a user payload

POST /auth/signin Login to account

Parameters

No parameters

Request body ^{required}

application/json

Examples:

[Modified value]

```
{
  "email": "nozdryakovbogdan3112@mail.ru",
  "password": "bGdghsIH748Lkkd??"
}
```

Execute Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/auth/signin' \
  -H 'accept: application/json' \
  -H 'content-type: application/json' \
  -d '{
    "email": "nozdryakovbogdan3112@mail.ru",
    "password": "bGdghsIH748Lkkd??"
  }'
```

Request URL

http://localhost:5000/auth/signin

Server response

Code	Details
200	<p>Response body</p> <pre>{ "status": 200, "data": { "name": "ingaale", "email": "nozdryakovbogdan3112@mail.ru" }, "message": "Authorization was successful" }</pre> <p>Response headers</p> <pre>access-control-allow-credentials: true access-control-allow-origin: * connection: keep-alive content-length: 121 content-type: application/json; charset=utf-8 date: Mon, 22 Jul 2024 04:19:26 GMT etag: W/"79-507Lf4rtzPN1pr4ANV/4ltx00" keep-alive: timeout=5 x-powered-by: Express</pre>

Мы успешно вошли в новый аккаунт. Теперь создадим новый пост с помощью метода POST `/api/posts/public`, в котором будет использоваться такое же ключевое слово, как в посте из предыдущего аккаунта:

posts

GET

/api/posts/public

Get all user posts

POST

/api/posts/public

Create a new post

Parameters

Cancel

Reset

No parameters

Request body required

application/json

Examples:

[Modified value]

```
{
  "title": "Я влюбился в IVE",
  "access": "public",
  "content": "Я влюбился в IVE! Они очень классные..."
}
```

Execute

Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:5000/api/posts/public' \
  -H 'accept: application/json' \
  -H 'content-type: application/json' \
  -d '{
    "title": "Я влюбился в IVE",
    "access": "public",
    "content": "Я влюбился в IVE! Они очень классные..."
  }'
```

Request URL

http://localhost:5000/api/posts/public

Server response

Code

Details

201

Response body

```
{
  "status": 201,
  "data": {
    "id": "81caa256-1b2a-4082-ab6f-2063cdfa6e9d",
    "title": "Я влюбился в IVE",
    "access": "public",
    "content": "Я влюбился в IVE! Они очень классные...",
    "rating": 0,
    "tags": null
  },
  "message": "Post successfully created"
}
```

Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 261
content-type: application/json; charset=utf-8
date: Mon, 22 Jul 2024 04:22:37 GMT
etag: W/"105-kjwKAsBYag35p/w3zXdkcyj5eM"
keep-alive: timeout=5
location: /api/posts/81caa256-1b2a-4082-ab6f-2063cdfa6e9d
x-powered-by: Express
```

Пост создался успешно. Теперь выполним полнотекстовый поиск через метод GET `/api/posts/public`, в который передадим то ключевое слово:

posts

GET

/api/posts/public

Get all user posts

Parameters

Cancel

Name	Description
searchParam string (query)	Search parameter that sets among whom the search substring will be searched - titles or contents <div>--</div>
searchSubstring string (query)	Search substring that is searched among titles or contents <div>влюб</div>

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/posts/public?searchSubstring=ND0xND2ND0xND0xND1ND0xND0xND1' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public?searchSubstring=ND0xND2ND0xND0xND1ND0xND0xND1
```

Server response

Code

Details

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "81ca256-1b2a-488f-ab6f-2063daf6e9d",
      "title": "Я влюбился в IVE",
      "access": "public",
      "content": "Я влюбился в IVE! Они очень классные...",
      "rating": 0,
      "tags": null
    }
  ],
  "message": "List of all posts"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 255
content-type: application/json; charset=utf-8
date: Mon, 22 Jul 2024 04:24:38 GMT
etag: W/"ff-AfJf8NVf1G54fKw1by0j+bnw00"
keep-alive: timeout=5
x-powered-by: Express
```

Метод вернул нам именно новый пост из нового аккаунта. Метод не стал нам возвращать пост из предыдущего аккаунта, даже несмотря на то, что и там, и там, при полнотекстовом поиске – использовалось одно и то же ключевое слово. Выполним этот же метод, но уже без полнотекстового поиска, чтобы получить все посты пользователя:

posts

GET

/api/posts/public

Get all user posts

Parameters

Cancel

Name	Description
searchParam string (query)	Search parameter that sets among whom the search substring will be searched - titles or contents <div>--</div>
searchSubstring string (query)	Search substring that is searched among titles or contents <div>searchSubstring</div>

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:5000/api/posts/public' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public
```

Server response

CodeDetails

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "1edfc759-927a-4bba-8337-efe74706531a",
      "title": "Властелин колец - книги против фильмов",
      "access": "public",
      "content": "Я прочитал все книги, а также посмотрел все фильмы про Властелин Колец...",
      "rating": 0,
      "tags": [
        "фильмы",
        "книги",
        "Властелин Колец"
      ]
    },
    {
      "id": "81caa256-1b2a-4882-ab6f-2063daf6e9d",
      "title": "Я влюбился в IVE",
      "access": "public",
      "content": "Я влюбился в IVE! Они очень классные...",
      "rating": 0,
      "tags": null
    }
  ],
  "message": "List of all posts"
}
```

Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 624
content-type: application/json; charset=utf-8
date: Mon, 22 Jul 2024 04:26:59 GMT
etag: W/"270-jZ34rd6/+saX0yJZ3Tj36+hJmZA"
keep-alive: timeout=5
x-powered-by: Express
```

Метод не вернул нам пост из предыдущего аккаунта. Значит, что все работает верно – пользователи могут получать только свои посты, а чужие нет.