

Стажировка

«Веб-приложение для публикации постов – поиск по фразам»

Выполнил: Ноздряков Богдан Валериевич

Проверил: Пармузин Александр Игоревич

Санкт-Петербург

2024

Условие:

Необходимо реализовать новый режим поиска для метода получения всех постов – поиск по фразам.

Если есть посты:

1. “кирпичная дорогая изящная стена”
2. “кирпичная дорогая стена”
3. “кирпичные стены”
4. “кирпичные дорогие стены”

По фильтру `?filters[title][phraseSearch]=”кирпичная стена”` – были найдены все посты, кроме первого, так как расстояние между запрашиваемыми словами не больше одного слова.

Анализ:

Текущая реализация полнотекстового поиска позволяет найти слово, или набор слов в документах. Но также пользователь может быть заинтересованным в том, чтобы искать именно фразу. Например, пользователь ищет что-то вроде “быстро прыгать” или “прыгать очень быстро”. Если искать с помощью текущей реализации – “прыгать & быстро”, то будут найдены документы, содержащие эти слова, но мы получим любую случайную конфигурацию внутри документа, независимо от того, связаны они синтаксически или нет.

Итак, нам нужно внедрить в существующую реализацию полнотекстового поиска “механизм”, который будет не просто искать документы, в которых одновременно содержатся все слова из запроса: “прыгать & быстро”, а одновременно содержатся все слова из запроса в определенном диапазоне. По заданию, этот диапазон – не больше одного слова, то есть между запрашиваемыми словами может либо вообще не быть других слов: “прыгать быстро”, либо может быть только одно слово: “прыгать очень быстро”. Таким образом, мы сохраним преимущество полнотекстового поиска в получении различных форм слова, а также увеличим точность результатов, выдав только те, где запрашиваемые слова находятся максимально рядом друг с другом (задается расстояние между запрашиваемыми словами). Именно в этом и заключается фразовый поиск в контексте полнотекстового поиска.

Решить данную задачу с помощью нативных средств PostgreSQL невозможно (если не пытаться сделать что-то с помощью регулярных выражений, использование которых не является наилучшим решением, так как регулярные выражения будет тяжело правильно внедрить, а также они будут сильно влиять на производительность). Поэтому придется воспользоваться чем-то сторонним, что поможет нам в решении данной задачи. Для этого воспользуемся поисковым движком Elasticsearch.

ElasticSearch:

ElasticSearch – это поисковая система, которая позволяет выполнять поиск документов более гибко, чем нативный PostgreSQL. Данный сервис предоставляет нереляционное хранилище данных, а также методы API, чтобы пользоваться данным хранилищем. Система позволяет работать с данными в формате JSON.

Разберемся подробнее что это все значит:

- 1) Реляционное хранилище данных – хранилище данных, где данные представляются в виде таблиц (система из строк и столбцов), а каждая запись (строка) в таблице связана с другими записями через ключи (таким образом таблицы связываются друг с другом). Здесь каждая запись (строка) имеет уникальный идентификатор (ключ), а столбцы предоставляют атрибуты данных. Данные хранилища используют язык запросов SQL для манипулирования данными. Поэтому такие хранилища называют SQL базы данных.
- 2) Нереляционное хранилище данных – хранилище данных, где для хранения применяется модель, которая оптимизирована для хранения определенного типа содержимого. Например, данные могут храниться в виде документов JSON, графов, а также ключ-значений. Такие хранилища не используют язык запросов SQL, и вместо него запросы осуществляются с помощью иных языков и конструкций. Поэтому такие хранилища называют NoSQL базы данных

Реляционные хранилища используют тогда, когда необходимо, чтобы база данных соответствовала требованиям ACID – атомарность, непротиворечивость, изолированность и долговечность. Это позволяет уменьшить вероятность неожиданного поведения системы и обеспечить целостность базы данных. Достигается подобное путем жесткого определения того, как именно транзакции взаимодействуют с базой данных. Также такие хранилища используются, когда данные, с которыми работают структурированы. То есть в реляционных хранилищах ставится в основу целостность данных.

Нереляционные хранилища используют, когда в основе гибкость и скорость, а не 100% целостность данных. Также нереляционные хранилища чаще используют, когда происходит работа с большими объемами данных из-за лучшей скорости работы с данными.

Разберем ACID:

- Атомарность – все изменения в рамках одной транзакции либо полностью применяются, либо вообще не применяются. Если происходит сбой во время выполнения транзакции, все изменения, сделанные в процессе этой транзакции, откатываются до состояния, которое было до начала транзакции. Это обеспечивает целостность данных, предотвращая появление несогласованных состояний в базе данных.

Примером может служить банковский перевод. Если перевод состоит из двух шагов: списание средств с одного счета и зачисление их на другой, оба этих действия должны быть выполнены успешно. Если один из шагов не удастся (например, из-за технических проблем), весь процесс откатывается, и средства остаются на исходном счете, сохраняя баланс счетов.

- Непротиворечивость - любая транзакция приводит базу данных из одного согласованного состояния в другое согласованное состояние. Все правила ограничений, установленные на данные (например, уникальность ключей, ограничения внешнего ключа), должны соблюдаться после завершения каждой транзакции. Это помогает избежать ситуации, когда данные находятся в неконсистентном состоянии.

Если, например, в базе данных есть правило, что каждый заказ должен иметь хотя бы одно связанное значение в таблице товаров, то попытка удалить последнее связанное значение для данного заказа должна привести к откату транзакции, чтобы сохранить целостность данных.

- Изолированность – гарантирует, что одновременно выполняемые транзакции не влияют друг на друга до тех пор, пока они не будут завершены. Это свойство позволяет транзакциям выполняться независимо друг от друга, даже если они работают с теми же данными. В результате, каждая транзакция видит базу данных в том состоянии, в котором она была в начале транзакции, исключая любые изменения, сделанные другими транзакциями, которые еще не были подтверждены.

Предположим, две транзакции одновременно пытаются увеличить счет пользователя на определенную сумму. Без изоляции одна транзакция могла бы увидеть промежуточное состояние счета, измененное другой транзакцией, что могло бы привести к неправильному итоговому значению счета. С помощью изоляции каждая транзакция видит фиксированное значение счета на протяжении всего своего выполнения, что обеспечивает корректность итогового результата.

- Долговечность - гарантирует, что после успешного завершения транзакции все изменения, внесенные в базу данных, сохраняются и остаются постоянными, даже в случае сбоя системы. Это означает, что даже если система перезагрузится или произойдет сбой, все транзакции, которые были успешно завершены до этого момента, будут сохранены в базе данных.

Если пользователь отправил платеж через интернет-банк, и эта транзакция была успешно завершена, но затем произошел сбой сервера перед тем, как информация о платеже была записана в базу данных, принцип долговечности гарантирует, что информация о платеже все равно будет сохранена и зарегистрирована в базе данных после восстановления системы.

Целостность данных – свойство базы данных, при котором всегда соблюдаются правила и ограничения, которые определяют допустимые состояния данных. Целостность данных важна для обеспечения точности и надежности информации, хранящейся в базе данных.

Например, если в базе данных есть ограничение, что возраст человека не может быть отрицательным числом, любая попытка вставить или обновить запись с отрицательным возрастом будет отклонена, чтобы поддерживать целостность данных.

Транзакция – это набор CRUD операций по работе с базой данных, объединенных в одну атомарную пачку. Чтобы обратиться к базе данных – нужно сначала открыть соединение с ней. Это называется коннект. Коннект – это просто труба, по которой мы посылаем запросы. Чтобы сгруппировать запросы в одну атомарную пачку, используем транзакцию. Транзакцию надо:

1. Открыть.
2. Выполнить все операции внутри.
3. Закрыть.

Как только мы закрыли транзакцию, труба освободилась. И ее можно переиспользовать, отправив следующую транзакцию.

Можно, конечно, каждый раз закрывать соединение с БД. И на каждое действие открывать новое. Но эффективнее переиспользовать текущие. Потому что создание нового коннекта — тяжелая операция, долгая. Каждая транзакция удовлетворяет ACID.

NoSQL базы позволяют выполнять то, чего не умеют SQL базы:

- База данных NoSQL хранит большие объемы неструктурированной информации. То есть такая база данных не накладывает ограничений на типы хранимых данных. Более того, при необходимости в процессе работы можно добавлять новые типы данных
- Использование облачных хранилищ — требует, чтобы данные можно было легко распределить между несколькими серверами для обеспечения масштабирования. Использование, для тестирования и разработки, локального оборудования, а затем перенос системы в облако, где она и работает — это именно то, для чего созданы NoSQL базы данных
- Нереляционные базы данных работают быстрее реляционных баз, так как они не используют язык запросов SQL для работы с данными, а используют другие конструкции и языки, которые позволяют работать с данными быстрее

Как было сказано ранее, Elasticsearch относится к категории NoSQL, то есть предоставляет нереляционное хранилище данных. Также, как было сказано ранее, в базу данных Elasticsearch данные с помощью API отправляются в виде документов JSON и хранятся в таком же виде:

```
{
  "id": "223d526d-5064-455e-9daf-6e7e3ad3e77d",
  "title": "Футбол в начале нулевых",
  "authorId": "93243b0e-6fbf-4a68-a6c1-6da4b4e3c3e4"
}
```

Такой выбор был обусловлен тем, что необходимо было добиться хорошей скорости поиска при очень больших данных, что как-раз и дают нереляционные хранилища. Все дело в том, что намного быстрее читать из объектов, содержащих все необходимое здесь и сейчас. И намного проще вносить изменения в неструктурированную схему данных.

ElasticSearch сохраняет документ в индекс кластера и делает его доступным для поиска. После этого можно найти и извлечь документ, используя API ElasticSearch.

Основой для работы с текстовыми документами является анализатор. Он представляет собой цепочку последовательных обработчиков. Сначала поступивший в анализатор текст проходит символьные фильтры, которые убирают, добавляют или заменяют отдельные символы в потоке. Например, с помощью символьных фильтров можно заменить арабские цифры (٠ ١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩) на современные арабские цифры (0123456789), перевести текст в нижний регистр или удалить HTML-теги.

Затем Elasticsearch передает обработанный текст токенизатору, который очищает поток символов от знаков препинания и разбивает по определенным правилам на отдельные слова — токены. В зависимости от токенизатора можно получить как набор слов, так и набор, где будут лишь основы слов (например, корни).

После токенизатора система передает набор слов в один или несколько фильтров токенов, которые могут добавлять, удалять или менять слова в наборе. Например, фильтр стоп-токенов может удалять часто используемые служебные слова, такие как артикли *a* и *the* в англоязычном тексте. На выходе из анализатора мы имеем набор токенов, который помещается в индекс. Это позволяет сохранять максимум смысла при минимуме знаков.

Elasticsearch ищет слова из запроса уже по индексу. При этом поисковые индексы можно разделить на сегменты — шарды. На каждом узле (запущенном экземпляре Elasticsearch) может быть размещено несколько сегментов. Каждый узел действует как координатор для делегирования операций правильному сегменту, а перебалансировка и маршрутизация выполняются автоматически.

Такие системы, как Elastic, могут использоваться как самостоятельные хранилища, так и в купе с базой данных. Когда база данных (например, Postgres) используется вместе с ElasticSearch, то подразумевается, что данные сначала ищутся в Elastic, а затем по найденным id извлекаются данные из базы данных Postgres, так как в ней хранится больше данных, чем в Elastic. На больших объемах это может быть полезно, так как поиск через движок происходит быстрее, однако нужна постоянная синхронизация, а данные дублируются и занимают вдвое больше места. Синхронизация подразумевает, что данные пишутся в основную базу данных Postgres и сразу же после этого — в базу данных ElasticSearch. С удалением также. Здесь могут быть проблемы. Если процесс оборвался на

середине, то поскольку нет поддержки транзакций – набор данных в двух системах начинает отличаться.

Индексы в Elasticsearch:

Ранее мы много упоминали индекс в контексте Elasticsearch. Однако как именно он работает и что это вообще такое:

Шард в Elasticsearch – это логическая единица хранения данных на уровне базы, которая содержит часть данных индекса. Шарды позволяют распределить данные по различным узлам в кластере для обеспечения масштабируемости и отказоустойчивости. Когда создается индекс, определяется количество первичных и реплицированных шардов. Первичные шарды отвечают за обработку всех операций записи и чтения, в то время как реплики используются для увеличения доступности данных и балансировки нагрузки. Все документы хранятся именно в шардах.

Индекс в Elasticsearch – это одновременно и распределенная база и механизм управления и организации данных, это именно логическое пространство. Индекс содержит один или более шардов, их совокупность и является хранилищем. То есть это место, где хранятся все данные. Индекс состоит из одного или нескольких шардов и используется для организации данных по схожести. Можно представить индекс как аналог таблицы в реляционной базе данных, где каждая строка представляет собой документ, а столбцы — поля документа.

Разберемся с кластером:

В Elasticsearch за операции поиска и индексации отвечает отдельный инстанс Lucene(*шард*). Для того, чтобы обращаться к распределенной системе шардов, нам необходимо иметь некий координирующий узел, именно он будет принимать запросы и давать задания на запись или получение данных. То есть помимо хранения данных мы выделяем еще один вариант поведения программы — координирование.

Каждый запущенный экземпляр **Elasticsearch** является отдельным узлом (node). **Cluster** — это совокупность определенных нод. Когда запускается один экземпляр - кластер будет состоять из одной ноды.

Таким образом мы изначально ориентируемся на два вида узлов — CRUD-узлы и координирующие узлы, но их может быть больше. У нас есть куча машин, объединенных в сеть и все это очень напоминает кластер. Можно сказать, что:

- Узел – это сервер (отдельный экземпляр Elasticsearch), который хранит данные и участвует в обработке запросов к этим данным – предоставляет API для работы с документами. Узлы могут быть разных типов, включая узлы данных (хранят и обрабатывают данные), координаторские узлы (координируют запросы между

клиентами и узлами данных), управляющие узлы (отвечают за управление кластером) и т.д.

- Кластер — это коллекция одного или нескольких узлов Elasticsearch, которые работают вместе для обеспечения индексации данных и выполнения запросов к этим данным. Кластеры обеспечивают высокую доступность и масштабируемость, позволяя распределение данных и нагрузки между узлами.

Преимущества Elasticsearch:

- Хорошее качество и скорость обработки текста
- Возможность сочетать в поиске как полнотекстовый поиск, так и фразовый поиск (учитываются расстояния между запрашиваемыми словами)
- Легкое управление
- Отказоустойчивость (при сбое одного из узлов индекс перераспределяется на оставшиеся узлы, используя внутренний механизм репликации данных. Кластеры Elasticsearch продолжают работать, даже если возникают аппаратные ошибки типа сбоя узла или неполадок сети)
- Высокая горизонтальная масштабируемость (система может быть запущена на десятках или сотнях узлов вместо одного мощного сервера, а добавить в кластер еще один узел можно практически без дополнительных настроек)

Недостатки Elasticsearch:

- Необходимость в постоянной синхронизации с основной базой данных
- Необходимость хранить данные, помимо основной базы данных, в базе данных Elastic. То есть данные занимают вдвое больше места
- Нет поддержки транзакций, из-за чего может нарушиться синхронизация с основной базой данных в случае какого-нибудь сбоя
- Использование клиента Elasticsearch может быть нелучшим решением, так как при построении сложных запросов – могут появиться огромные нечитаемые циклы

ElasticSearch – нативный PostgreSQL:

Используя нативный PostgreSQL – мы получим ограниченную функциональность. В некоторых случаях ее будет достаточно, но не всегда (например, нативный PostgreSQL не сможет реализовать фразовый поиск, который требуется реализовать по текущему заданию). И как-раз в случаях, когда такой функциональности будет недостаточно – на помощь приходит Elasticsearch.

Решение:

Настройка Docker:

Как было описано ранее, заданный фразовый поиск будет реализован с помощью поискового движка Elasticsearch. Чтобы не устанавливать Elasticsearch на свой компьютер – воспользуемся Docker для запуска экземпляра Elasticsearch в Docker-контейнере. Мы воспользуемся docker-compose для развертывания экземпляра Elasticsearch. Для этого создадим файл docker-compose.yaml:

```
🐙 docker-compose.yaml
1  version: '3.6'
2  services:
3    elasticsearch:
4      container_name: elasticsearch_container
5      image: elasticsearch:8.14.1
6      volumes:
7        - esdata:/usr/share/elasticsearch/data
8      environment:
9        - discovery.type=single-node
10       - xpack.security.enabled=false
11      logging:
12        driver: none
13      ports:
14        - 9200:9200
15        - 9300:9300
16      networks:
17        - esnet
18  volumes:
19    esdata:
20  networks:
21    esnet:
```

Разберем данный файл:

- version '3.6' – указываем версию используемого docker-compose
- в services мы определяем все сервисы, то есть контейнеры, которые будут запущены. В данном случае – будет запущен только один контейнер elasticsearch
- в elasticsearch мы указываем параметры для данного контейнера
- container_name – указываем имя данного контейнера, которое отображается в списке запущенных контейнеров

- `image` – образ Docker, из которого будет создан контейнер. В данном случае – образ ElasticSearch версии 8.14.1
- `volumes` – задаем тома, которые будут монтированы в контейнер. В данном случае монтируется том `esdata`, использующийся для хранения данных ElasticSearch (если не создавать тома, то после перезагрузки Docker – все данные из ElasticSearch будут потеряны)
- `environment` – устанавливаем переменные окружения в контейнер. Здесь `discovery.type=single-node` – устанавливаем тип обнаружения как одноузловый; `xpack.security.enabled=false` – так как нам не нужно иметь дело с SSL, поскольку приложение запускается локально (данная конфигурация подходит только для локальной разработки. Для производственного развертывания – нужны будут другие параметры)
- `logging` – задаем настройки логирования. В данном случае – мы отказываемся от вывода логов в какой-нибудь файл или консоль
- `ports` – определяем порты, которые будут проброшены из контейнера на хост-машину. В данном случае мы пробрасываем порты 9200 и 9300, которые будут использоваться для доступа к ElasticSearch
- `networks` – указываем сеть, в которой должен работать контейнер. В данном случае – это сеть `esnet`

После создания `docker-compose.yml`, мы можем запустить экземпляр ElasticSearch в Docker через команду `docker-compose up`. Для этого внесем в `package.json` новый скрипт:

```
"start:compose": "docker-compose up",
```

Теперь, перед запуском приложения, нужно запустить `npm run start:compose` для запуска экземпляра ElasticSearch.

Внедрение ElasticSearch в приложение:

После настройки Docker, можно переходить к написанию функционала, который позволит нам обращаться к базе данных ElasticSearch. Но для начала – отменим миграцию, которую мы создали при реализации полнотекстового поиска нативными средствами PostgreSQL (напомню, что эта миграция создавала колонку `text_tsvector` типа `tsvector` в таблице `Post`, создавала триггер для обновления `text_tsvector` и создавала индекс GIN). Для этого создадим новую миграцию (`./src/database/migrations/main/post/delete/20240804103201-delete_text_tsvector_column_from_post_table.js`), которая будет отменять данную:

```

src > database > migrations > main > post > delete > JS 20240804103201-delete_text_tsv_column_from_post_table.js > ...
1  async function up({ context: queryInterface }) {
2      const tablePostInfo = await queryInterface.describeTable('Posts');
3      const tablePostIndexes = await queryInterface.showIndex('Posts');
4      const tablePostGinIndex = tablePostIndexes.find(index => index.name === 'post_title_content_gin_idx');
5
6      if (tablePostGinIndex) {
7          await queryInterface.removeIndex('Posts', 'post_title_content_gin_idx');
8      }
9
10     if (tablePostInfo.text_tsv) {
11         await queryInterface.removeColumn('Posts', 'text_tsv');
12     }
13
14     await queryInterface.sequelize.query(`DROP TRIGGER IF EXISTS update_text_tsv_trigger ON "Posts"`);
15     await queryInterface.sequelize.query(`DROP FUNCTION IF EXISTS update_text_tsv`);
16     await queryInterface.sequelize.query(`
17         DELETE FROM "SequelizeMeta" WHERE name = '20240719151942-add_text_tsv_column_to_post_table.js';
18     `);
19 }
20 module.exports = { up }
21

```

Здесь мы удаляем все, что было создано той миграцией, делая проверку на их наличие:

- Индекс GIN post_title_content_gin_idx
- Столбец text_tsv
- Триггер update_text_tsv_trigger
- Функция update_text_tsv

Также мы удаляем саму миграцию 20240719151942-add_text_tsv_column_to_post_table.js из таблицы SequelizeMeta, которая хранит все миграции.

Создадим новый класс ElasticSearchProvider

(./src/database/elasticsearch/elasticsearch.provider.ts), который будет провайдером к клиенту ElasticSearch, что позволит нам работать с базой данных ElasticSearch:

src > database > elasticsearch > TS elasticsearch.provider.ts > ElasticSearchProvider > populateIndex

```
1  import { Client } from '@elastic/elasticsearch';
2  import {
3    SearchRequest,
4    SearchResponse,
5    BulkRequest,
6    IndicesCreateRequest
7  } from '@elastic/elasticsearch/lib/api/types';
8
9  class ElasticSearchProvider {
10    private readonly client: Client;
11
12    constructor() {
13      this.client = new Client({
14        node: process.env.ELASTIC_URL_DEV
15      });
16    }
17
18    public async isIndexExist(indexName: string): Promise<boolean> {
19      return (
20        await this.client.indices.exists({
21          index: indexName
22        })
23      );
24    }
25
26    public async createIndex(indexSettings: IndicesCreateRequest): Promise<void> {
27      await this.client.indices.create(indexSettings);
28    }
```

```
30    public async populateIndex(docs: BulkRequest[]): Promise<void> {
31      await this.client.bulk({
32        refresh: true,
33        operations: docs
34      });
35    }
36
37    public async deleteIndex(indexName: string): Promise<void> {
38      await this.client.indices.delete({
39        index: indexName
40      });
41    }
42
43    public async indexDocument(indexName: string, documentId: string, documentBody: object): Promise<void> {
44      await this.client.index({
45        index: indexName,
46        id: documentId,
47        document: documentBody
48      });
49    }
```

```

51     public async updateDocument(
52         indexName: string,
53         documentId: string,
54         newDocumentBody: object
55     ): Promise<void>
56     {
57         await this.client.update({
58             index: indexName,
59             id: documentId,
60             doc: newDocumentBody
61         });
62     }
63
64     public async deleteDocument(indexName: string, documentId: string): Promise<void> {
65         await this.client.delete({
66             index: indexName,
67             id: documentId
68         });
69     }
70
71     public async searchByRequest(searchRequest: SearchRequest): Promise<SearchResponse> {
72         return (
73             await this.client.search(searchRequest)
74         );
75     }
76 }
77 export default ElasticSearchProvider;

```

Разберем данный класс:

- В конструкторе мы создаем клиент ElasticSearch через подключение к серверу ElasticSearch по его URL (определяем URL в переменной ELASTIC_URL_DEV файла env). Данный клиент позволит нам работать с базой данных ElasticSearch
- isIndexExist – метод, который через созданный клиент, проверяет – существует ли заданный индекс в базе данных ElasticSearch
- createIndex - метод, который через созданный клиент, создает индекс в базе данных ElasticSearch через переданную конфигурацию индекса
- populateIndex - метод, который через созданный клиент, заполняет индекс множеством переданных документов в базе данных ElasticSearch
- deleteIndex - метод, который через созданный клиент, удаляет указанный индекс в базе данных ElasticSearch
- indexDocument - метод, который через созданный клиент, добавляет в указанный индекс один документ в базе данных ElasticSearch
- updateDocument - метод, который через созданный клиент, обновляет указанный документ в указанном индексе базы данных ElasticSearch

- `deleteDocument` - метод, который через созданный клиент, удаляет указанный документ в указанном индексе базы данных `ElasticSearch`
- `searchByRequest` - метод, который через созданный клиент, производит поиск в базе данных `ElasticSearch`

Также не забудем создать модуль `ElasticSearchModule` (`./src/database/elasticsearch/elasticsearch.module.ts`), который будет создавать синглтон `ElasticSearchProvider` и делать его доступным для инъектирования:

```
src > database > elasticsearch > TS elasticsearch.module.ts > ...
1  import dependencyContainer from '../utils/lib/dependencyInjection/dependency.container';
2  import ElasticSearchProvider from './elasticsearch.provider';
3
4  class ElasticSearchModule {
5      constructor() {
6          dependencyContainer.registerInstance('esProvider', new ElasticSearchProvider());
7      }
8  }
9  export default ElasticSearchModule;
```

Также нужно создать сам объект `ElasticSearchModule` в модуле базы данных `DatabaseModule` (`./src/database/database.module.ts`):

```
dependencyContainer.registerInstance('esModule', new ElasticSearchModule());
```

После создания провайдера `ElasticSearch`, нужно написать функцию, которая создаст индекс в базе данных `ElasticSearch` и добавит все существующие посты в этот индекс (напомню, что данный поиск применяется только к постам пользователя). Благодаря этому – мы сможем производить поиск документов в базе данных `ElasticSearch`. Так как это единичное действие, то создадим миграцию (`./src/database/migrations/elasticsearch/add/20240804144443-add_post_index_to_elasticsearch.js`), которая будет за это отвечать:

```
src > database > migrations > elasticsearch > add > JS 20240804144443-add_post_index_to_elasticsearch.js > up > settings
1  const { Client } = require('@elastic/elasticsearch');
2  const client = new Client({ node: 'http://localhost:9200' });
3  const postIndex = 'post_idx';
4
5  async function up({ context: queryInterface }) {
6      const isPostIndexExist = await client.indices.exists({
7          index: postIndex
8      });
9
10     if (isPostIndexExist) {
11         await client.indices.delete({
12             index: postIndex
13         });
14     }
15
16     const docs = [];
17     let posts = await queryInterface.sequelize.query(`
18         SELECT id, title, content, "authorId"
19         FROM "Posts"
20     `);
21
22     posts = posts[0];
```

```
24     await client.indices.create({
25         index: postIndex,
26         settings: {
27             max_ngram_diff: 12,
28             analysis: {
29                 filter: {
30                     russian_stop: {
31                         type: 'stop',
32                         stopwords: '_russian_'
33                     },
34                     russian_stemmer: {
35                         type: 'stemmer',
36                         language: 'russian'
37                     },
38                     ngram_filter: {
39                         type: 'edge_ngram',
40                         min_gram: 3,
41                         max_gram: 15
42                     }
43                 },
```

```
44     analyzer: {
45         russian_custom_analyzer: {
46             type: 'custom',
47             tokenizer: 'standard',
48             filter: [
49                 'lowercase',
50                 'trim',
51                 'russian_stop',
52                 'russian_stemmer',
53                 'ngram_filter'
54             ]
55         }
56     }
57 }
58 },
```



```
58 mappings: {
59     properties: {
60         id: {
61             type: 'keyword'
62         },
63         title: {
64             type: 'text',
65             analyzer: 'russian_custom_analyzer',
66             search_analyzer: 'russian'
67         },
68         content: {
69             type: 'text',
70             analyzer: 'russian_custom_analyzer',
71             search_analyzer: 'russian'
72         },
73         authorId: {
74             type: 'keyword'
75         }
76     }
77 }
78 });
```

```

80     posts.forEach(post => {
81         docs.push({
82             index: {
83                 _index: postIndex,
84                 _id: post.id
85             }
86         });
87         docs.push(post);
88     });
89
90     await client.bulk({
91         refresh: true,
92         operations: docs
93     });
94 }
95
96 async function down({ context: queryInterface }) {
97     const isPostIndexExist = await client.indices.exists({
98         index: postIndex
99     });
100
101     if (isPostIndexExist) {
102         await client.indices.delete({
103             index: postIndex
104         });
105     }
106 }
107
108 module.exports = { up, down }

```

Разберем данную миграцию подробно. Сначала мы подключаемся к базе данных Elasticsearch через клиент, а также указывает имя индекса, который будет создан:

```

1  const { Client } = require('@elastic/elasticsearch');
2  const client = new Client({ node: 'http://localhost:9200' });
3  const postIndex = 'post_idx';

```

Дальше определена функция `up`. В ней проверяется – существует ли уже в базе данных Elasticsearch указанный индекс. Если он существует, то в таком случае – он удаляется. Затем определяется массив `docs`, который будет заполнен документами для заполнения ими индекса. Также в `posts` мы получаем все документы, которые будем хранить в индексе. В нашем случае – это объекты, представляющие посты пользователей (объекты будут хранить частичную информацию о постах – `id`, `title`, `content`, `authorId`. Такой выбор будет подробно объяснен чуть позже):

```
5  async function up({ context: queryInterface }) {
6      const isPostIndexExist = await client.indices.exists({
7          index: postIndex
8      });
9
10     if (isPostIndexExist) {
11         await client.indices.delete({
12             index: postIndex
13         });
14     }
15
16     const docs = [];
17     let posts = await queryInterface.sequelize.query(`
18         SELECT id, title, content, "authorId"
19         FROM "Posts"
20     `);
21
22     posts = posts[0];
```

После создается сам индекс:

```
24     await client.indices.create({
25         index: postIndex,
26         settings: {
27             max_ngram_diff: 12,
28             analysis: {
29                 filter: {
30                     russian_stop: {
31                         type: 'stop',
32                         stopwords: '_russian_'
33                     },
34                     russian_stemmer: {
35                         type: 'stemmer',
36                         language: 'russian'
37                     },
38                     ngram_filter: {
39                         type: 'edge_ngram',
40                         min_gram: 3,
41                         max_gram: 15
42                     }
43                 },
```

```
44     analyzer: {
45         russian_custom_analyzer: {
46             type: 'custom',
47             tokenizer: 'standard',
48             filter: [
49                 'lowercase',
50                 'trim',
51                 'russian_stop',
52                 'russian_stemmer',
53                 'ngram_filter'
54             ]
55         }
56     }
57 }
58 },
```

```

58     mappings: {
59         properties: {
60             id: {
61                 type: 'keyword'
62             },
63             title: {
64                 type: 'text',
65                 analyzer: 'russian_custom_analyzer',
66                 search_analyzer: 'russian'
67             },
68             content: {
69                 type: 'text',
70                 analyzer: 'russian_custom_analyzer',
71                 search_analyzer: 'russian'
72             },
73             authorId: {
74                 type: 'keyword'
75             }
76         }
77     }
78 });

```

В поле `index` – мы указываем индекс, который создаем. Далее в `settings` устанавливается конфигурация данного индекса:

- `max_ngram_diff: 12` – указываем, что максимальная разница между минимальным граммом и максимальным равна 12 (`max_gram – min_gram`; такой выбор будет подробно объяснен позже)
- `analysis` – определяем конфигурацию анализа документов при индексировании:
 - `filter` – настраиваем фильтры, которые будут применены к токенам:
 - `russian_stop` – фильтр, который удаляет все стоп-слова из токенов, учитывая именно русский язык
 - `russian_stemmer` – фильтр, который производит стемминг (сокращение каждого слова к его корневой форме (стему)), учитывая именно русский язык

- `ngram_filter` – фильтр, который будет преобразовывать токены к n-граммам (в данном случае – к `edge_ngram`)
- `analyzer` – определяем конфигурацию анализаторов документов:
 - `russian_custom_analyzer` – настройка кастомного анализатора:
 - `type: 'custom'` – параметр, указывающий, что анализатор – кастомный
 - `tokenizer: 'standard'` – параметр, указывающий, что каждый документ в индексе будет разбиваться на токены через стандартный токенайзер (разделяет текст на термины по границам слов, как определено алгоритмом сегментации текста в Юникоде. Он удаляет большинство символов препинания)
 - `filter` – параметр, который определяет фильтры, которые должны быть применены к каждому токenu:
 1. `lowercase` – фильтр, приводящий каждый токен к нижнему регистру
 2. `trim` – фильтр, удаляющий пробелы в начале и в конце токена (если они там присутствуют)
 3. `russian_stop` – параметр, применяющий определенный в поле `filter - russian_stop` фильтр
 4. `russian_stemmer` - параметр, применяющий определенный в поле `filter - russian_stemmer` фильтр
 5. `ngram_filter` - параметр, применяющий определенный в поле `filter - ngram_filter` фильтр

В `mappings` определяется структура документов, которые будут храниться в данном индексе. В данном случае – каждый документ будет содержать:

- `id` своего поста
- `title` своего поста
- `content` своего поста
- `authorId` своего поста

`-title` и `content` мы добавили, так как сам поиск осуществляется именно по столбцам `title` и `content` постов;

-id мы добавили, чтобы пользователь после применения поиска, видел id каждого найденного поста и мог найти определенный интересующий его пост через id этого поста (документ в базе данных Elasticsearch содержит не все данные поста. Пользователь может захотеть просмотреть дополнительные данные поста);

-authorId мы добавили, чтобы было возможно выполнять поиск именно для постов, которые принадлежат текущему пользователю (обычный пользователь может искать по поиску только свои посты).

-Мы не заложили в документ других данных поста, так как это не требуется. Мы добавили только те данные, которые действительно нужны.

Для title и content мы установили тип text, так как именно такой тип имеют строки в базе данных Elasticsearch. Для id и authorId мы установили тип keyword – это специальный тип в базе данных Elasticsearch, который подходит для идентификаторов (если вдруг будет выполняться поиск по такой колонке, то там будет происходить поиск точного совпадения без учета морфологии слов или их порядка).

Также для title и content для индексирования мы установили определенный в settings анализатор: 'russian_custom_analyzer'. Это нужно, так как наш поиск должен не только учитывать расстояние между запрашиваемыми словами. Поиск должен еще выполнять полнотекстовый поиск, то есть учитывать регистры и разные формы слова. Так как все посты – на русском языке, мы установили специальный кастомный анализатор russian_custom_analyzer, который будет решать данный вопрос. Без установления такого анализатора – поиск будет не способен выполнять поиск по разным словоформам. Также для title и content мы установили анализатор 'russian' для запросов (такой выбор будет подробно объяснен позже).

После создания индекса, мы все полученные существующие посты добавляем в созданный индекс. Для этого мы ранее определили специальный массив docs и сейчас добавляем туда все посты следующим образом:


```

81     posts.forEach(post => {
82         docs.push({
83             index: {
84                 _index: postIndex,
85                 _id: post.id
86             }
87         });
88         docs.push(post);
89     });
90
91     await client.bulk({
92         refresh: true,
93         operations: docs
94     });

```

Сначала добавляется специальный объект, который содержит имя индекса и его id (которое будет равно id текущего поста):

```

docs.push({
    index: {
        _index: postIndex,
        _id: post.id
    }
});

```

Имя индекса мы добавляем, чтобы было понятно – в какой индекс добавляется документ. А id мы добавляем, чтобы в последствии можно было обращаться к документу индекса по id этого документа (если самому не указать id документа при его добавлении в индекс, то Elasticsearch автоматически добавит свой id документу. Мы не будем знать этот id. Соответственно, когда мы захотим получить какой-то документ из базы данных Elasticsearch, например, чтобы обновить его данные, или удалить его, мы не сможем его получить, так как мы не знаем его id. А когда мы указываем документу id, равный id поста документа, при его добавлении в индекс, мы легко сможем получить нужный нам документ, так как будем знать его id).

После мы добавляем объект, содержащий данные поста (id, title, content, authorId):

```
docs.push({
  id: post.id,
  title: post.title,
  content: post.content,
  authorId: post.authorId
});
```

И так для каждого существующего поста. Мы перед каждым постом в массиве добавляем тот объект конфигурации, чтобы указать методу, который будет добавлять данный пост в индекс, что мы хотим именно добавить в этот заданный индекс следующий элемент в массиве (то есть пост), а также установить ему данный id. После того, как все посты и все конфигурации под все посты будут добавлены в массив, мы все это добавим в созданный индекс через метод bulk клиента Elasticsearch.

То есть данная конфигурация будет работать следующим образом:

-Будет создан индекс post_idx под посты пользователя в базе данных Elasticsearch. Индекс будет хранить документы, представляющие посты пользователей, а именно объекты, содержащие id, title, content, authorId определенного поста. К title и content при индексации будет применен кастомный анализатор russian_custom_analyzer. Это означает, что каждый раз, когда в индекс будет попадать документ, для его полей title и content будет применена следующая процедура:

1. Весь текст будет разбиваться на отдельные слова (токены), удаляя знаки препинания
2. К каждому полученному слову (токену) будет применены фильтры:
 - Первый фильтр приведет каждый токен к нижнему регистру (чтобы при пользовательском поиске – поиск не был чувствителен к регистру)
 - Второй фильтр удалит пробелы в начале и конце каждого токена, если они там присутствуют (дополнительная мера безопасности, чтобы в случае чего – поиск не учитывал никаких пробелов)
 - Третий фильтр удалит токены, которые являются стоп-словами, так как они не несут никакой полезной информации, и поиск, по таким словам, производить не нужно
 - Четвертый фильтр произведет стемминг (приведение слова к его корню) для каждого токена (нам абсолютно не нужно будет учитывать окончания слов при поиске, поэтому есть

смысл уменьшать размер слов, убирая их суффикс и окончание)

- Пятый фильтр создаст для каждого токена `edge_ngram`

Разберем подробно что такое `ngram` и `edge_ngram`, а также зачем мы привели каждый токен из `title` и `content` к типу `edge_ngram`:

`ngram` – это тип токенизации для обработки текста. Данный тип разбивает слова на последовательности символов (граммы) фиксированной длины `n`. Например, если `min_ngram: 3`, `max_ngram: 6` и задано слово ‘кирпич’, то данный тип следующим образом разобьет слово ‘кирпич’:

‘кир’, ‘ирп’, ‘рпи’, ‘пич’
‘кирп’, ‘ирпи’, ‘рпич’,
‘кирпи’, ‘ирпич’
‘кирпич’

`edge_ngram` – это тип токенизации для обработки текста. Данный тип работает также, как `ngram`, но он разбивает слова на граммы именно с начала слова. Например, если `min_ngram: 3`, `max_ngram: 6` и задано слово ‘кирпич’, то данный тип следующим образом разобьет слово ‘кирпич’:

‘кир’
‘кирп’
‘кирпи’
‘кирпич’

Итак, мы сделали так, что при добавлении в индекс документа, его `title` и его `content` превращаются в `edge_ngram`, чтобы повысить точность поиска. Все дело в том, что если использовать стандартный русский анализатор при индексации, то он в некоторых случаях будет вычислять неверный корень для токенов, оставляя их суффиксы. Например, такой стандартный русский анализатор для слова ‘кирпичный’ вычислит корень – ‘кирпичн’. Соответственно, когда пользователь попытается что-то найти по запросу ‘кирпич’ – он не получит тот документ со словом ‘кирпичный’, так как анализатор установил для него основу не ‘кирпич’, а ‘кирпичн’. А пользователь может не догадаться, что нужно вводить именно ‘кирпичн’, а не ‘кирпич’. Соответственно, стандартный русский анализатор – плохо работает со словоформами, что сильно ухудшит поиск.

Именно по этой причине – чтобы улучшить качество поиска за счет учета правильных словоформ, мы и приводим столбцы поиска к типу `edge_ngram` при индексации, так как данный тип, как было сказано ранее, разбивает слово на разные последовательности букв, из которых состоит данное слово. При правильной конфигурации `edge_ngram` (`min_ngram` и `max_ngram`) – мы обязательно будем в определенном грамме получать корень данного слова. Мы выбрали `min_ngram = 3`, а `max_ngram = 15`, так как это будет покрывать большинство слов, по которым будет проходить поиск. Нельзя в `min_ngram` указывать 1, так как таких слов, состоящих из одной буквы – нет. Также в `min_ngram` нельзя указывать 2, так как слова, состоящие из 2 букв –

это стоп-слова, которые не имеют никакой логической нагрузки. А вот слова, состоящие из трех букв – это уже могут быть полноценные слова. Именно поэтому мы установили `min_gram = 3`. А `max_gram` равен именно 15, так как это будет покрывать большинство слов, которые пользователь будет искать. Конечно, в русском языке есть слова, длина которых больше 15. Но данные слова довольно редко употребляются. Это значит, что пользователь довольно редко будет искать данные слова. И если пользователь будет искать слова, длина которых больше 15 – как-раз стемминг анализатора запроса уменьшит размер слова. И с учетом стемминга размер слова станет либо равным 15, либо меньше 15. В таком случае поиск пройдет штатно. Конечно, слово может быть очень большим, и даже с учетом стемминга – его длина будет больше 15. В таком случае – пользователю придется указывать в запросе только часть этого слова, а не все слово. Но тут выбора нет – мы не можем поставить сильно большой `max_gram`, так как это повлияет на размер индекса и производительность поиска. Как я сказал ранее, ситуации, когда в слове после стемминга длина по-прежнему будет больше 15 – очень сильно редки. Параметр `max_gram = 15` будет покрывать практически всегда все слова в поиске.

Был выбран тип `edge_ngram`, а не `ngram`, так как `edge_ngram` предоставляет более точный результат за счет того, что в нем все граммы строятся с начала слова. Так в разы улучшается поиск. Вернемся, к примеру `ngram` с `min_gram: 3`, `max_gram: 6`, ‘кирпич’:

‘кир’, ‘ирп’, ‘рпи’, ‘пич’
‘кирп’, ‘ирпи’, ‘рпич’,
‘кирпи’, ‘ирпич’
‘кирпич’

`ngram` и `edge_gram` работают таким образом, что они ищут запрос в своих граммах. Если хоть один грамм будет совпадать с запросом, то документ будет возвращен. Получается, если мы используем `ngram`, пользователь производит поиск по запросу ‘кирпич’, то будут возвращены все документы, содержащие хоть один данный грамм:

‘кир’, ‘ирп’, ‘рпи’, ‘пич’
‘кирп’, ‘ирпи’, ‘рпич’,
‘кирпи’, ‘ирпич’
‘кирпич’

Среди таких документов, помимо тех, которые будут содержать словоформы ‘кирпич’, также будут содержаться документы, которые содержат, например, следующие слова:

‘Типичный’ (содержит грамм ‘пич’, который также содержится в слове ‘кирпич’),
‘Ирпине’ (содержит грамм ‘ирп’, который также содержится в слове ‘кирпич’).

Тип `edge_ngram` содержит в разы меньше граммов и все они создаются с начала слова:

‘кир’
‘кирп’
‘кирпи’
‘кирпич’

Именно поэтому мы выбрали `edge_ngram`, а не `ngram` - `edge_ngram` в разы увеличит точность всех документов, которые будут получены в результате поиска, так как он строит меньшее количество граммов, и они более точны.

Так работает анализатор для `title` и `content` именно при индексировании (добавлении документа в индекс). Однако мы в конфигурации индекса для `title` и `content` также указали: `'search_analyzer': 'russian'`

Это означает, что запрос поиска, который будет проходит по полям `title` или `content` – будет обрабатываться стандартным русским анализатором.

Однако зачем мы использовали разные анализаторы для индексирования и запроса? Все дело в том, что по умолчанию, если в конфигурации не указать `'search_analyzer'`, то для запроса будет применен такой же анализатор, как и для индексирования. У нас анализатор индексирования – создает из слов `edge_ngram`. Значит и анализатор запроса – будет создавать из запроса `edge_ngram`. Как я уже сказал ранее - `edge_ngram` и `ngram` работают так, что они ищут хотя бы одно совпадениеграмма. Это значит, что даже `edge_ngram`, хоть он и в разы точнее `ngram` – будет выдавать, помимо подходящих запросов, неподходящие запросы. Вернемся, к примеру разбиения 'кирпич' на граммы через `edge_ngram` с `min_gram: 3`, `max_gram: 6`:

'кир'
'кирп'
'кирпи'
'кирпич'

Здесь, помимограмма 'кирпич', много других граммов. Это означает, что при запросе 'кирпич' – мы также будем получать несоответствующие запросы. К примеру, у нас есть следующие посты:

- 'Мой кирпичный дом'
- 'Моя кирка'

Когда пользователь вводит 'кирпич' – он рассчитывает получить только пост:

- 'Мой кирпичный дом'

Однако пользователь получит оба этих поста, так как слово 'кирка' в посте 'Моя кирка' – содержитграмм 'кир'. А этот жеграмм содержится и в слове 'кирпич'.

Именно по этой причине – мы используем разные анализаторы для индексации и запроса. Анализатор `edge_ngram` используется при индексации, чтобы поиск работал с правильными словоформами. А стандартный русский анализатор используется для

запроса, чтобы сделать поиск по `edge_ngram` – более точным, чтобы он выдавал только те документы, которые действительно соответствуют запросу. Все дело в том, что стандартный русский анализатор приведет запрос к нижнему регистру, удалит стоп-слова и произведет стемминг (приведение слова к его корню). Таким образом, в запросе мы получим всего лишь один точный грамм для каждого слова, который и будет представлять корень данного слова. И мы просто будем искать среди всех граммов один точный грамм – корень слова. А если бы мы в запросе использовали также анализатор `edge_ngram`, то в запросе мы бы получали не один точный грамм, а много. И поэтому мы бы получали результаты по множеству граммов, то есть таким образом, к результатам, соответствующим запросу – мы бы подмешивали результаты, несоответствующие запросу.

Тут важно сказать, что если бы мы использовали анализатор `ngram` для индексирования и стандартный русский анализатор для запроса, то это также правильно бы работало, так как мы бы также искали только один определенный грамм за счет другого анализатора в запросе. Но все-таки было решено взять именно `edge_ngram`, так как он предоставляет меньшее количество граммов, а также они более точные. Нам здесь просто нет смысла расширять размер индекса большим количеством неточных граммов, которые предоставляет `ngram`.

Таким образом, мы создадим в базе данных Elasticsearch индекс под посты пользователя, а также добавим в этот индекс все существующие на данный момент посты. Теперь после запуска приложения – один раз применится данная миграция, и будет не нужно каждый раз при запуске проверять наличие данного индекса, что создать его в случае отсутствия.

Внедрим функционал, позволяющий выполнять фразовый поиск. Для этого расширим функционал класса `PostService` (`./src/domain/post/post.service.ts`). Сначала добавим переменную, которая содержит индекс постов в базе данных Elasticsearch, а также инжектируем провайдер `ElasticSearchProvider`:

```
class PostService extends DomainService {
  private readonly esIndex: string = 'post_idx';
  private readonly esProvider: ElasticSearchProvider;
```

```
  constructor(esProvider: ElasticSearchProvider) {
    super();
    this.esProvider = esProvider;
  }
```

Определим метод, позволяющий получать индекс постов:

```
public getEsIndex(): string {  
    return this.esIndex;  
}
```

После создадим новый метод phraseSearch:

```
83     public async phraseSearch(  
84         user: IUserPayload,  
85         searchParam: 'title' | 'content',  
86         searchString: string  
87     ): Promise<unknown[]>  
88     {  
89         const matchPhrase = {  
90             [searchParam]: {  
91                 query: searchString,  
92                 slop: 1  
93             }  
94         };
```

```

96  ✓    const phraseSearchSettings: IPhraseSearch = {
97  ✓      admin: {
98          index: this.esIndex,
99  ✓      query: {
100         match_phrase: matchPhrase
101       }
102     },
103  ✓    user: {
104         index: this.esIndex,
105  ✓    query: {
106  ✓      bool: {
107  ✓        must: [
108  ✓          {
109             match_phrase: matchPhrase
110           },
111  ✓          {
112  ✓            term: {
113                 authorId: user.id
114             }
115           }
116        ]
117      }
118    },
119    _source_excludes: [ 'authorId' ]
120  }
121  };

```

```

122
123    const searchResult = await this.esProvider.searchByRequest(phraseSearchSettings[user.role]);
124    return searchResult.hits.hits.map(hit => hit._source);
125  }

```

Разберем данный метод. Метод принимает три параметра:

- user – данные текущего пользователя, который выполняет фразовый поиск
- searchParam – параметр, по которому осуществляется фразовый поиск (либо поле title (заголовки постов), либо поле content (текст постов))
- searchString – строка, по которой пользователь осуществляет фразовый поиск

Далее создаются конфигураторы – `matchPhrase` и `phraseSearchSettings`, на основе которых строится запрос фразового поиска. Конфигуратор `phraseSearchSettings` содержит конфигурацию поиска для админов и для обычных пользователей.

Рассмотрим конфигурацию для админов:

```
admin: {
  index: this.esIndex,
  query: {
    match_phrase: matchPhrase
  }
},
```

-Здесь указывается индекс, в котором нужно вести поиск (в данном случае – это индекс постов в базе данных Elasticsearch, который мы ранее определили в поле `esIndex`). Также здесь мы описываем сам запрос в поле `query`. В данном случае – используется `match_phrase` запрос – тип запроса, который используется для поиска документов, содержащих указанную фразу. Данный запрос создается на основе определенного ранее конфигулятора `matchPhrase`:

```
const matchPhrase = {
  [searchParam]: {
    query: searchString,
    slop: 1
  }
};
```

-Этот конфигулятор устанавливает поле, по которому будет происходить `match_phrase` запрос. В данном случае – это то поле, которое было передано в функцию в параметре `searchParam`. Также данный конфигулятор говорит, что поиск должен проходить по переданной в функцию строке `searchString` (`query: searchString`), а также между запрашиваемыми словами в этой строке либо вообще нет других слов, либо есть только одно слово (`slop: 1`).

-Таким образом, мы получим `match_phrase` запрос, который будет искать указанную строку, где слова в указанной строке должны идти в том же порядке, в котором определены в указанной строке, а также расстояние между запрашиваемыми словами – не более одного слова.

Теперь рассмотрим конфигурацию для обычных пользователей:

```
user: {
  index: this.esIndex,
  query: {
    bool: {
      must: [
        {
          match_phrase: matchPhrase
        },
        {
          term: {
            authorId: user.id
          }
        }
      ]
    }
  },
  _source_excludes: [ 'authorId' ]
}
```

-Здесь также указывается индекс, в котором нужно вести поиск (в данном случае — это индекс постов в базе данных Elasticsearch, который мы ранее определили в поле `esIndex`). Также создается запрос на основе критериев, определенных в `query`. Мы указали `bool`-запрос, чтобы можно было выполнить несколько критериев поиска, а `must` является как-раз массивом всех условий, которые нужно выполнить. В данном случае здесь два условия:

- Используется такой же `match_phrase` запрос, как в конфигурации для админов
- Используется `term` запрос, который ищет документы, содержащие точное значение для указанного поля. В данном случае — ищем документы, где `authorId` равен `id` текущего пользователя, который в данный момент выполняет фразовый поиск. Благодаря этому — пользователь может выполнять фразовый поиск только среди своих постов, и не может получить посты другого пользователя

Также мы указываем `_source_excludes` — это параметр, который указывает поля, которые следует исключить из возвращаемых документов. В данном случае, мы из всех полученных результатов исключаем поле `authorId`, так как обычный пользователь и так получит в результате только свои посты, и ему не нужно знать свой собственный `id` в системе.

В конце мы выполняем, созданный конфигураторами, запрос к базе данных Elasticsearch через описанного ранее провайдера ElasticsearchProvider:

```
const searchResult = await this.esProvider.searchByRequest(phraseSearchSettings[user.role]);
```

То есть мы получаем из конфигуратора phraseSearchSettings соответствующий запрос для фразового поиска, передав в него роль текущего пользователя.

Далее мы просто извлекаем документы из полученного ответа от Elasticsearch и возвращаем массив полученных документов:

```
return searchResult.hits.hits.map(hit => hit._source);
```

Добавим еще один метод wordSearch в PostService:

```
83     public async wordSearch(  
84         user: IUserPayload,  
85         searchParam: 'title' | 'content',  
86         searchString: string  
87     ): Promise<unknown[]> {  
88         const queryMust = searchString.split(' ').map(word => ({  
89             match: {  
90                 [searchParam]: word  
91             }  
92         }));
```

```

94     const wordSearchSettings: IPostSearch = {
95       admin: {
96         index: this.esIndex,
97         query: {
98           bool: {
99             must: [
100               ...queryMust
101             ]
102           }
103         }
104       },
105       user: {
106         index: this.esIndex,
107         query: {
108           bool: {
109             must: [
110               ...queryMust,
111               {
112                 term: {
113                   authorId: user.id
114                 }
115               }
116             ]
117           }
118         },
119         _source_excludes: ['authorId']
120       }
121     };

```

```

122
123     const searchResult = await this.esProvider.searchByRequest(wordSearchSettings[user.role]);
124     return searchResult.hits.hits.map(hit => hit._source);
125   }

```

Если предыдущий метод `phraseSearch` отвечал за фразовый поиск, то данный метод отвечает за обычный полнотекстовый поиск, так как у пользователя есть выбор: либо использовать фразовый поиск, либо обычный полнотекстовый поиск.

Данный метод работает по такому же принципу, как и `phraseSearch`. Отличается он только конфигурацией поиска, которая настроена не под фразовый, а под полнотекстовый поиск. Отличие от конфигурации `phraseSearch` в том, что здесь строится не `match_phrase` запрос, а просто запрос `match`, который просто ищет совпадения. Данный запрос разобьет строку поиска по пробелам, а потом произведет одновременный поиск каждого слова в определенном поле. Таким образом, мы получим обычный полнотекстовый поиск.

Не будем забывать, что пользователь может создать новый пост, изменить существующий, удалить старый. Теперь все эти изменения нужно вносить также и в базу данных `ElasticSearch`. Для этого используем `ElasticSearchProvider`:

- В методе `createPost` будем добавлять новый созданный пост не только в нашу базу данных, но и в базу данных `ElasticSearch`:

```
public async createUserPost(user: IUserPayload, postDataCreate: IPostCreate): Promise<Post> {
    const newPost = await Post.create({
        ...postDataCreate
    });

    await this.esProvider.indexDocument(this.esIndex, newPost.id, {
        id: newPost.id,
        title: newPost.title,
        content: newPost.content,
        authorId: newPost.authorId
    });

    return (
        await this.getPostByUniqueParams({
            where: {
                id: newPost.id
            }
        }, user)
    );
}
```

- В методе `updatePost` будем проверять — обновляет ли пользователь `title` и `content` поста. Если обновляет, то обновим эти данные в базе данных `ElasticSearch`:

```

173     public async updateUserPost(post: Post, newPostData: IPostUpdate): Promise<Post> {
174         if (newPostData.title) {
175             await this.esProvider.updateDocument(this.esIndex, post.id, {
176                 title: newPostData.title
177             });
178         }
179
180         if (newPostData.content) {
181             await this.esProvider.updateDocument(this.esIndex, post.id, {
182                 content: newPostData.content
183             });
184         }
185
186         Object.assign(post, newPostData);
187         await post.save();
188         return post;
189     }

```

- В методе deletePost будем удалять пост не только в нашей базе данных, но и в базе данных ElasticSearch:

```

public async deleteUserPost(post: Post): Promise<void> {
    await this.esProvider.deleteDocument(this.esIndex, post.id);
    await post.destroy();
}

```

Изменим документацию Swagger (./src/middleware/swagger/swagger.yaml) так, чтобы пользователь при поиске постов мог использовать специальный фильтр, который будет определять параметр поиска, тип поиска и запрос:

```

462     /api/posts/public:
463         get:
464             summary: Get all user posts
465             tags:
466                 - posts
467             security:
468                 - BearerAuth: []
469             parameters:
470                 - in: query
471                   name: filters
472                   schema:
473                       type: object
474                       properties:
475                           title || content:
476                               type: object
477                               properties:
478                                   phraseSearch || wordSearch:
479                                       type: string
480             required: false
481             description: Object that defines filters for searching among titles or contents
482             style: deepObject

```

Теперь Swagger будет позволять пользователю при поиске постов - применять фильтр к поиску. Данный фильтр будет представлять такой объект:

```
{
  "title || content": {
    "wordSearch || phraseSearch": string
  }
}
```

То есть сначала мы указываем – по какому полю производить поиск (либо title, либо content). Далее мы указываем поиск, который желаем применить (либо wordSearch (простой полнотекстовый поиск), либо phraseSearch (фразовый поиск)). Также мы указываем строку запроса, по которой будет проходить поиск. Данный фильтр сможет принимать одно из следующих значений:

1. Первый вариант:

```
{
  "title": {
    "wordSearch": string
  }
}
```

Означает – полнотекстовый поиск по полю title

2. Второй вариант:

```
{
  "title": {
    "phraseSearch": string
  }
}
```

Означает – фразовый поиск по полю title

3. Третий вариант:

```
{  
  "content": {  
    "wordSearch": string  
  }  
}
```

Означает – полнотекстовый поиск по полю content

4. Четвертый вариант:

```
{  
  "content": {  
    "phraseSearch": string  
  }  
}
```

Означает – фразовый поиск по полю content

Теперь пользователь будет отправлять специальный фильтр при поиске, который будет говорить – как производить поиск.

Изменим контроллер PostController (./src/domain/post/post.controller.ts), так как именно он работает с PostService. В данном контроллере за фразовый поиск отвечает метод getAllPosts. Значит его и изменим:


```

17 class PostController {
18     private readonly postFiltersSettings: IPostFiltersSettings = {
19         search: {
20             params: ['title', 'content'],
21             methods: [
22                 {
23                     name: 'wordSearch',
24                     method: (
25                         user: IUserPayload,
26                         searchParam: 'title' | 'content',
27                         searchString: string
28                     ) => this.postService.wordSearch(user, searchParam, searchString)
29                 },
30                 {
31                     name: 'phraseSearch',
32                     method: (
33                         user: IUserPayload,
34                         searchParam: 'title' | 'content',
35                         searchString: string
36                     ) => this.postService.phraseSearch(user, searchParam, searchString)
37                 }
38             ]
39         }
40     };

```

```

46 @JoiRequestValidation({
47     type: 'query',
48     name: 'filters'
49 }, PostFiltersSchema)
50 public async getAllUserPosts(req: Request, res: Response, next: NextFunction): Promise<Response | void> {
51     try {
52         const user = req.user as IUserPayload;
53         const filters = req.query.filters as IPostFilters;
54         let posts = [];
55
56         if (!filters) {
57             posts = await this.postService.getAllUserPosts({}, user);
58         }

```

```

59     else {
60         const searchParam = this.postFiltersSettings.search.params.find(
61             param => param in filters
62         ) as ('title' | 'content');
63
64         const searchParamBody: ISearchType = JSON.parse(filters[searchParam] as string);
65         const validError = joiValidation(searchParamBody, PostSearchSchema);
66
67         if (validError) {
68             return res.status(422).send(validError);
69         }
70
71         const searchMethod = this.postFiltersSettings.search.methods.find(
72             param => param.name in searchParamBody
73         ) as ISearchMethod;
74
75         const searchString = searchParamBody[searchMethod.name] as string;
76         posts = await searchMethod.method(user, searchParam, searchString);
77     }
78
79     return res.status(200).json({ status: 200, data: posts, message: "List of all posts" });
80 }
81 catch (err) {
82     next(err);
83 }
84 }

```

Разберем изменения:

- Так как пользователь теперь присылает фильтр, которые необходимо обработать – мы создали специальное поле, которое будет отвечать за обработку фильтра (с помощью данного поля, мы будем извлекать из фильтра – параметр, по которому будет проходить поиск, а также сам метод поиска):

```

18     private readonly postFiltersSettings: IPostFiltersSettings = {
19         search: {
20             params: ['title', 'content'],
21             methods: [
22                 {
23                     name: 'wordSearch',
24                     method: (
25                         user: IUserPayload,
26                         searchParam: 'title' | 'content',
27                         searchString: string
28                     ) => this.postService.wordSearch(user, searchParam, searchString)
29                 },
30                 {
31                     name: 'phraseSearch',
32                     method: (
33                         user: IUserPayload,
34                         searchParam: 'title' | 'content',
35                         searchString: string
36                     ) => this.postService.phraseSearch(user, searchParam, searchString)
37                 }
38             ]
39         }
40     };

```

- Мы проводим валидацию фильтра с помощью декоратора JoiRequestValidation (/src/validation/joi/decorator/joi.validation.decorator.ts):

```
@JoiRequestValidation({
  type: 'query',
  name: 'filters'
}, PostFiltersSchema)
```

Валидацию мы проводим по схеме PostFiltersSchema (/src/domain/post/validation/schema/post.search.schema.ts):

```
src > domain > post > validation > schema > TS post.search.schema.ts > ...
1  import Joi from 'joi';
2
3  export const PostFiltersSchema = Joi.alternatives().try(
4    Joi.object({
5      title: Joi.string().required()
6    }),
7    Joi.object({
8      content: Joi.string().required()
9    })
10 ).optional();
11
12 export const PostSearchSchema = Joi.alternatives().try(
13   Joi.object({
14     wordSearch: Joi.string().required()
15   }),
16   Joi.object({
17     phraseSearch: Joi.string().required()
18   })
19 ).required();
20
```

Схема PostFiltersSchema проверяет, что в фильтре содержится либо свойство 'title', либо свойство 'content'. Ничего другого быть не должно.

- Мы получаем данные пользователя, который проводит поиск, фильтр, который пользователь отправил, а также определяем массив постов:

```
const user = req.user as IUserPayload;
const filters = req.query.filters as IPostFilters;
let posts = [];
```

- Если фильтр не был задан, то выполняется стандартный поиск всех существующих постов пользователя:

```
if (!filters) {
  posts = await this.postService.getAllUserPosts({}, user);
}
```

- Если же фильтр задан:

```
else {
  const searchParam = this.postFiltersSettings.search.params.find(
    param => param in filters
  ) as ('title' | 'content');

  const searchParamBody: ISearchType = JSON.parse(filters[searchParam] as string);
  const validError = joiValidation(searchParamBody, PostSearchSchema);

  if (validError) {
    return res.status(422).send(validError);
  }

  const searchMethod = this.postFiltersSettings.search.methods.find(
    param => param.name in searchParamBody
  ) as ISearchMethod;

  const searchString = searchParamBody[searchMethod.name] as string;
  posts = await searchMethod.method(user, searchParam, searchString);
}
```

В таком случае — мы начинаем обрабатывать полученный фильтр. Чтобы произвести поиск — нам нужно понять по какому полю мы должны его производить, какой поиск применять, а также какой сам запрос. Именно здесь мы это выясняем:

1. В searchParam мы получаем из фильтра поле, по которому должен проходить поиск. В нашем случае — это либо 'title', либо 'content'
2. В searchParamBody — мы получаем тело параметра в фильтре. То есть как мы помним — фильтр имеет такой тип:

```
{  
  "title || content": {  
    "wordSearch || phraseSearch": string  
  }  
}
```

Когда пользователь отправит такой фильтр – свойство ‘title’ или ‘content’ будет содержать именно строку, а не объект. Именно поэтому нам нужно получить тело параметра:

```
{  
  "wordSearch || phraseSearch": string  
}
```

Распарсить это тело из строки в нужный тип с помощью `JSON.parse`. Именно это мы и выполняем в `searchParamBody`.

Дальше мы валидируем распарсенное тело с помощью функции валидатора `joi` – `joiValidation` (`./src/validation/joi/joi.validation.ts`). В качестве схемы – мы используем `PostSearchSchema` (`./src/domain/post/validation/schema/post.search.schema.ts`):

```

src > domain > post > validation > schema > TS post.search.schema.ts > ...
1  import Joi from 'joi';
2
3  export const PostFiltersSchema = Joi.alternatives().try(
4    Joi.object({
5      title: Joi.string().required()
6    }),
7    Joi.object({
8      content: Joi.string().required()
9    })
10 ).optional();
11
12 export const PostSearchSchema = Joi.alternatives().try(
13   Joi.object({
14     wordSearch: Joi.string().required()
15   }),
16   Joi.object({
17     phraseSearch: Joi.string().required()
18   })
19 ).required();
20

```

Данная схема проверяет, что тело запроса – это либо поле “wordSearch”, содержащее строковое значение, либо поле “phraseSearch”, содержащее строковое значение.

3. Если валидация прошла правильно, то есть пользователь отправил правильный фильтр, то дальше в searchMethod мы извлекаем из фильтра метод, которым хочет воспользоваться пользователь для поиска – либо wordSearch (простой полнотекстовый поиск), либо phraseSearch (фразовый поиск)
4. В searchString мы извлекаем из фильтра строку запроса
5. С помощью созданного ранее поля postFiltersSettings, мы вызываем нужный метод поиска из сервиса PostService, и записываем результат в массив posts
6. В конце возвращаем результат пользователю:

```

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

```

Тестирование:




Запуск экземпляра Elasticsearch в контейнере Docker:

Сначала запустим контейнер Elasticsearch. Для этого выполним команду `npm run start:compose` в корне нашего приложения:

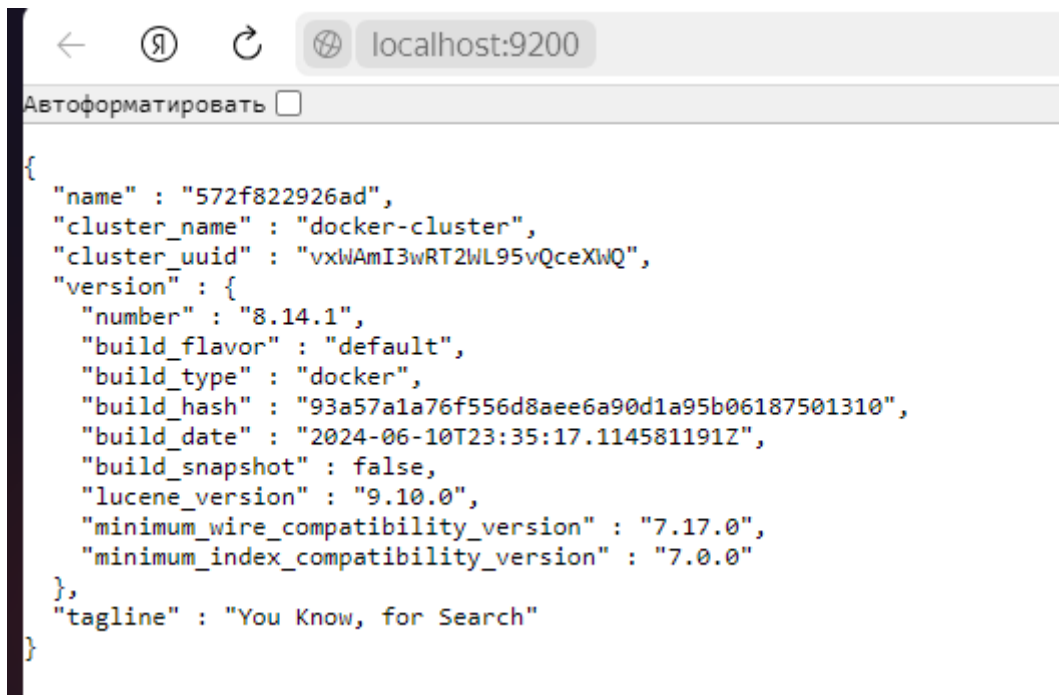
```
PS C:\Users\Богдан\Desktop\Консорцеум Кодекс\Стажировка\daccord> npm run start:compose
```

```
time="2024-07-30T19:27:15+03:00" level=warning msg="C:\Users\Богдан\Desktop\Консорцеум Кодекс\Стажировка\daccord\docker-compose.yml: 'version' is obsolete"
[+] Running 1/0
  ✓ Container elasticsearch_container Created 0.0s
Attaching to elasticsearch_container
elasticsearch_container | Jul 30, 2024 4:27:19 PM sun.util.locale.provider.LocaleProviderAdapter <clinit>
elasticsearch_container | WARNING: COMPAT locale provider will be removed in a future release
elasticsearch_container | {"@timestamp":"2024-07-30T16:27:20.319Z", "log.level": "INFO", "message":"Using [jdk] n
elasticsearch_container | ative provider and native methods for [Linux]", "ecs.version": "1.2.0", "service.name":"ES_ECS", "event.dataset":"el
lasticsearch.server", "process.thread.name":"main", "log.logger":"org.elasticsearch.nativeaccess.NativeAccess", "elast
icsearch.node.name":"572f822926ad", "elasticsearch.cluster.name":"docker-cluster"}
elasticsearch_container | {"@timestamp":"2024-07-30T16:27:20.663Z", "log.level": "INFO", "message":"Java vector i
ncubator API enabled; uses preferredBitSize=256; FMA enabled", "ecs.version": "1.2.0", "service.name":"ES_ECS", "eve
nt.dataset":"elasticsearch.server", "process.thread.name":"main", "log.logger":"org.apache.lucene.internal.vectoriza
tion.PanamaVectorizationProvider", "elasticsearch.node.name":"572f822926ad", "elasticsearch.cluster.name":"docker-cl
uster"}
```

```
QfOqpqvjAFwnXw", "elasticsearch.node.name":"572f822926ad", "elasticsearch.cluster.name":"docker-cluster")
elasticsearch_container | {"@timestamp":"2024-07-30T16:27:32.081Z", "log.level": "INFO", "message":"Node [{572f82
2926ad}{cmHfNz6KQfOqpqvjAFwnXw}] is selected as the current health node.", "ecs.version": "1.2.0", "service.name":"
ES_ECS", "event.dataset":"elasticsearch.server", "process.thread.name":"elasticsearch[572f822926ad][management][T#3]
", "log.logger":"org.elasticsearch.health.node.selection.HealthNodeTaskExecutor", "elasticsearch.cluster.uuid":"vxWA
mI3wRT2WL95vQceXWQ", "elasticsearch.node.id":"cmHfNz6KQfOqpqvjAFwnXw", "elasticsearch.node.name":"572f822926ad", "ela
sticsearch.cluster.name":"docker-cluster"}
elasticsearch_container | {"@timestamp":"2024-07-30T16:27:32.118Z", "log.level": "INFO", "current.health":"YELLOW
W", "message":"Cluster health status changed from [RED] to [YELLOW] (reason: [shards started [[post_idx][0]]]).", "p
revious.health":"RED", "reason":"shards started [[post_idx][0]]", "ecs.version": "1.2.0", "service.name":"ES_ECS", "
event.dataset":"elasticsearch.server", "process.thread.name":"elasticsearch[572f822926ad][masterService#updateTask]
[T#1]", "log.logger":"org.elasticsearch.cluster.routing.allocation.AllocationService", "elasticsearch.cluster.uuid":
"vxWAmI3wRT2WL95vQceXWQ", "elasticsearch.node.id":"cmHfNz6KQfOqpqvjAFwnXw", "elasticsearch.node.name":"572f822926ad"
, "elasticsearch.cluster.name":"docker-cluster"}
```

 View in Docker Desktop  View Config  Enable Watch

Проверим, что порт 9200 отвечает:

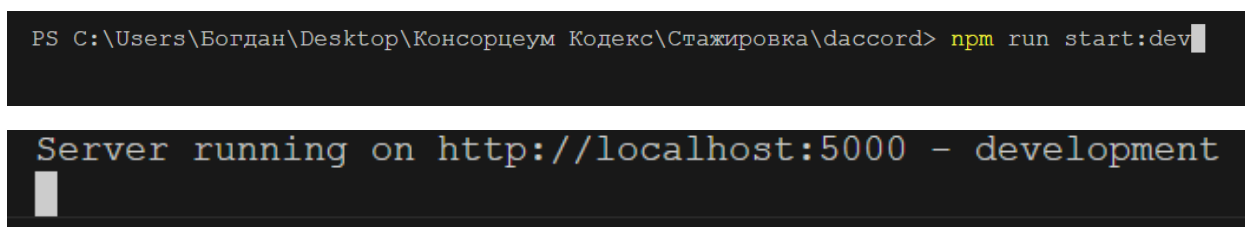


```
{
  "name" : "572f822926ad",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "vxWAmI3wRT2WL95vQceXWQ",
  "version" : {
    "number" : "8.14.1",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "93a57a1a76f556d8aee6a90d1a95b06187501310",
    "build_date" : "2024-06-10T23:35:17.114581191Z",
    "build_snapshot" : false,
    "lucene_version" : "9.10.0",
    "minimum_wire_compatibility_version" : "7.17.0",
    "minimum_index_compatibility_version" : "7.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

Экземпляр Elasticsearch в контейнере Docker был успешно запущен.

Запуск приложения:

После запуска сервера Elasticsearch – можно запустить само приложение. Для этого откроем в корне проекта новый терминал и введем туда `npm run start:dev`:



```
PS C:\Users\Вордан\Desktop\Консорцеум Кодекс\Стажировка\daccord> npm run start:dev

Server running on http://localhost:5000 - development
```

Сервер был успешно запущен.

Дальше проведем серию тестов через Swagger (<http://localhost:5000/api>).

Авторизация:

Сначала авторизируемся в системе:

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** /auth/signin (with a sub-label "Login to account")
- Parameters:** No parameters are listed.
- Request body:** Required, set to application/json.
- Examples:** A dropdown menu shows "[Modified value]". Below it, a JSON object is displayed:

```
{  "email": "awesom-e4000@mail.ru",  "password": "idnGj7dHq?!!!"}
```
- Buttons:** "Execute" (blue) and "Clear" (grey) buttons are at the bottom.
- Responses:** A section at the bottom for viewing the response.

The screenshot shows the response details from the REST client:

- Response body:** A JSON object is displayed:

```
{  "status": 200,  "data": {    "name": "jason_statham",    "email": "awesom-e4000@mail.ru"  },  "message": "Authorization was successful"}
```
- Response headers:** A list of headers is shown:

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 118
content-type: application/json; charset=utf-8
date: Tue, 30 Jul 2024 16:37:38 GMT
etag: W/"76-kfCoMat0X1/nDuSn4D1uf31D9So"
keep-alive: timeout=5
x-powered-by: Express
```

Мы успешно авторизовались в системе как обычный пользователь.

Получение всех постов пользователя:

Сначала получим все текущие посты пользователя (чтобы понимать далее при поиске, какие посты мы должны получать). Для этого выполним метод GET /api/posts/public без передачи фильтра:

posts

GET /api/posts/public Get all user posts

Parameters

Name

Description

filters

Object that defines filters for searching among titles or contents

object

(query)

Execute

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

Code

Details

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "fcb0acc1-1b41-4580-bee8-bd0b47f1f8e2",
      "title": "Моя кирпичная стена",
      "access": "public",
      "content": "Мой деревянный стул не выдержал...",
      "rating": 0,
      "tags": [
        "дом",
        "строительство",
        "ремонт"
      ]
    },
    {
      "id": "92b2b7b8-7337-4168-bef4-7c4249803f4f",
      "title": "Моя кирпичная дорогая стена",
      "access": "public",
      "content": "Поговорим об...",
      "rating": 0,
      "tags": null
    },
    {
      "id": "03fb773d-f2f2-4b29-bcc9-826e9b45b52c",
      "title": "Моя кирпичная дорогая измятая стена",
      "access": "public",
      "content": "Хочу рассказать об..."
    }
  ]
}
```

Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 2922
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 18:49:44 GMT
etag: W/"b6a-GXzcb7oK00hp191jbu15ys9URU"
keep-alive: timeout=5
x-powered-by: Express
```

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

Code	Details
------	---------

200

Response body

```
{
  "id": "03fb773d-f2f2-4b29-bcc9-826e9b45b52c",
  "title": "Моя кирпичная дорогая изваянная стена",
  "access": "public",
  "content": "Хочу рассказать об...",
  "rating": 0,
  "tags": null
},
{
  "id": "52d391e7-545c-42ae-968f-e0d74b93217a",
  "title": "Моя кирпичные дорожке изваянная стена",
  "access": "public",
  "content": "Мой дом меня очень радует...",
  "rating": 0,
  "tags": null
},
{
  "id": "ca4df99e-b896-40f5-80ff-73c8cea610bf",
  "title": "Моя кирпичные дорожке стены",
  "access": "public",
  "content": "Недавно я сделал очень хороший ремонт...",
  "rating": 0,
  "tags": null
},
{
  "id": "027cfc25-84fc-4e26-92d6-335ced596fe4",
  "title": "Моя кирпичные стены",
  "access": "public",
  "rating": 0,
  "tags": null
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 2922
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 10:49:44 GMT
etag: W/"b6a-GXzchrJokMOhp19Ijbu15ys9URU"
keep-alive: timeout=5
x-powered-by: Express
```

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

Code	Details
------	---------

200

Response body

```
{
  "id": "027cfc25-84fc-4e26-92d6-335ced596fe4",
  "title": "Моя кирпичные стены",
  "access": "public",
  "content": "Буду честен, я сделал...",
  "rating": 0,
  "tags": null
},
{
  "id": "a50ca562-14b5-446b-836a-5d56b5de3a8e",
  "title": "Кирпичные стены",
  "access": "public",
  "content": "Я долго думал...",
  "rating": 0,
  "tags": null
},
{
  "id": "b0933f7f-7b25-40d3-9f8d-212c160a2ecd",
  "title": "Стены кирпичные",
  "access": "public",
  "content": "Мой дом очень хорошо...",
  "rating": 0,
  "tags": null
},
{
  "id": "95a3293c-0ead-4858-80bb-c95b3551ba15",
  "title": "Новый Гарри Поттер",
  "access": "public",
  "rating": 0,
  "tags": null
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 2922
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 10:49:44 GMT
etag: W/"b6a-GXzchrJokMOhp19Ijbu15ys9URU"
keep-alive: timeout=5
x-powered-by: Express
```

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

Code Details

200

Response body

```
{
  "id": "95a3293c-0ead-4858-80bb-c95b3551ba15",
  "title": "Новый Гарри Поттер",
  "access": "public",
  "content": "Я хочу поделиться одной супер новостью...",
  "rating": 0,
  "tags": null
},
{
  "id": "da9ef067-74f5-4e73-b7a3-5a9364db5cc0",
  "title": "Хэсон из NPIX - лучшаа",
  "access": "public",
  "content": "Недавно я познакомился с очень интересной группой - NPIX. Участница группы Хэсон - просто супер!",
  "rating": 0,
  "tags": null
},
{
  "id": "44150a1f-d02b-4542-9c8a-afc8b5856cd0",
  "title": "Ремонт дома",
  "access": "public",
  "content": "Мой кирпичный дом...",
  "rating": 0,
  "tags": null
},
{
  "id": "aaaa3f48-08cd-4c26-b980-f56654a69d3a",
  "title": "Очень интересная история",
  "access": "public",
  "rating": 0,
  "tags": null
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 2922
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 10:49:44 GMT
etag: W/"b6a-GXzcbzJokU0hp19iJbu15ys9URU"
keep-alive: timeout=5
x-powered-by: Express
```

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

Code Details

200

Response body

```
{
  "id": "44150a1f-d02b-4542-9c8a-afc8b5856cd0",
  "title": "Ремонт дома",
  "access": "public",
  "content": "Мой кирпичный дом...",
  "rating": 0,
  "tags": null
},
{
  "id": "aaaa3f48-08cd-4c26-b980-f56654a69d3a",
  "title": "Очень интересная история",
  "access": "public",
  "content": "Я хочу сказать, что мой кирпич сегодня...",
  "rating": 0,
  "tags": null
},
{
  "id": "f1ea9135-86d2-4229-bbf3-efb1b09f6c23",
  "title": "Шахматы",
  "access": "public",
  "content": "Спустя долгое время, мне удалось добыть камень. Это несчастная кирка быстро сломалась...",
  "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: 2922
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 10:49:44 GMT
etag: W/"b6a-GXzcbzJokU0hp19iJbu15ys9URU"
keep-alive: timeout=5
x-powered-by: Express
```

Таким образом, мы получили следующие посты:

```
{
  "id": "fcb0acc1-1b41-4580-bee8-bd0b47f1f8e2",
  "title": "Моя кирпичная стена",
  "access": "public",
  "content": "Мой деревянный стул не выдержал...",
  "rating": 0,
  "tags": [
    "дом",
    "строительство",
    "ремонт"
  ]
},
{
  "id": "92b2b7b8-7337-4160-bef4-7c4249003f4f",
  "title": "Моя кирпичная дорогая стена",
  "access": "public",
  "content": "Поговорим об...",
  "rating": 0,
  "tags": null
},
{
  "id": "03fb773d-f2f2-4b29-bcc9-826e9b45b52c",
  "title": "Моя кирпичная дорогая изящная стена",
  "access": "public",
  "content": "Хочу рассказать об...",
  "rating": 0,
  "tags": null
},
{
  "id": "52d391e7-545c-42ae-968f-e0d74b93217a",
  "title": "Мои кирпичные дорогие изящные стены",
```

```
"access": "public",
"content": "Мой дом меня очень радует...",
"rating": 0,
"tags": null
},
{
  "id": "ca4df99e-b896-40f5-80ff-73c8cea610bf",
  "title": "Мои кирпичные дорогие стены",
  "access": "public",
  "content": "Недавно я сделал очень хороший ремонт...",
  "rating": 0,
  "tags": null
},
{
  "id": "027cfc25-84fc-4e26-92d6-335ced596fe4",
  "title": "Мои кирпичные стены",
  "access": "public",
  "content": "Буду честен, я сделал...",
  "rating": 0,
  "tags": null
},
{
  "id": "a50ca562-14b5-446b-836a-5d56b5de3a8e",
  "title": "Кирпичные стены",
  "access": "public",
  "content": "Я долго думал...",
  "rating": 0,
  "tags": null
},
{
  "id": "b0933f7f-7b25-40d3-9f8d-212c160a2ecd",
  "title": "Стены кирпичные",
```

```
"access": "public",
"content": "Мой дом очень хорошо...",
"rating": 0,
"tags": null
},
{
  "id": "95a3293c-0ead-4858-80bb-c95b3551ba15",
  "title": "Новый Гарри Поттер",
  "access": "public",
  "content": "Я хочу поделиться одной супер новостью...",
  "rating": 0,
  "tags": null
},
{
  "id": "da9ef067-74f5-4e73-b7a3-5a9364db5cc0",
  "title": "Хэвон из NMIXX - лучшая",
  "access": "public",
  "content": "Недавно я познакомился с очень интересной группой - NMIXX. Участница группы Хэвон - просто супер!",
  "rating": 0,
  "tags": null
},
{
  "id": "44150a1f-d02b-4542-9c8a-afc8b5856cd0",
  "title": "Ремонт дома",
  "access": "public",
  "content": "Мой кирпичный дом...",
  "rating": 0,
  "tags": null
},
{
  "id": "aaaa3f48-08cd-4c26-b980-f56654a69d3a",
  "title": "Очень интересная история",
```

```

    "access": "public",
    "content": "Я хочу сказать, что мой кирпич сегодня...",
    "rating": 0,
    "tags": null
  },
  {
    "id": "f1ea9135-86d2-4229-bbf3-efb1b09f6e23",
    "title": "Шахты",
    "access": "public",
    "content": "Спустя долгое время, мне удалось добыть камень. Это несчастная кирка быстро сломалась...",
    "rating": 0,
    "tags": null
  }
}

```

Именно на эти посты мы будем ориентироваться при поиске.

Фразовый поиск по title:

Выполним фразовый поиск фразы ‘кирпичная стена’ по столбцу title. Исходя из существующих постов пользователя, нам должны вернуться следующие посты:

```

{
  "id": "fcb0acc1-1b41-4580-bee8-bd0b47f1f8e2",
  "title": "Моя кирпичная стена",
  "access": "public",
  "content": "Мой деревянный стул не выдержал...",
  "rating": 0,
  "tags": [
    "дом",
    "строительство",
    "ремонт"
  ]
},

```



```
{
  "id": "92b2b7b8-7337-4160-bef4-7c4249003f4f",
  "title": "Моя кирпичная дорогая стена",
  "access": "public",
  "content": "Поговорим об...",
  "rating": 0,
  "tags": null
},
{
  "id": "ca4df99e-b896-40f5-80ff-73c8cea610bf",
  "title": "Мои кирпичные дорогие стены",
  "access": "public",
  "content": "Недавно я сделал очень хороший ремонт...",
  "rating": 0,
  "tags": null
},
{
  "id": "027cfc25-84fc-4e26-92d6-335ced596fe4",
  "title": "Мои кирпичные стены",
  "access": "public",
  "content": "Буду честен, я сделал...",
  "rating": 0,
  "tags": null
},
{
  "id": "a50ca562-14b5-446b-836a-5d56b5de3a8e",
  "title": "Кирпичные стены",
  "access": "public",
  "content": "Я долго думал...",
  "rating": 0,
  "tags": null
}
```

Для этого выполним GET /api/posts/public, передав в качестве параметра поиска – title, а в качестве метода поиска – phraseSearch, указав в строке запроса 'Кирпичная стена':

posts

GET

/api/posts/public

Get all user posts

⌵

Parameters

Cancel

Name	Description
filters	Object that defines filters for searching among titles or contents
object (query)	<pre>{ "title": { "phraseSearch": "Кирпичная стена" } }</pre>

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:5000/api/posts/public?filters$5$&title$50-$7$B$22phraseSearch$22$3A$22$D$9A$D$9B$8D1$80$D$0$B$7$D$9B$8D1$87$D$0$B$D$D$D$D$D$1$8F$2$D$D1$81$D1$82$D$0$B$5$D$0A$D$D$D$9B$22$7D' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public?filters$5$&title$50-$7$B$22phraseSearch$22$3A$22$D$9A$D$9B$8D1$80$D$0$B$7$D$9B$8D1$87$D$0$B$D$D$D$D$D$1$8F$2$D$D1$81$D1$82$D$0$B$5$D$0A$D$D$D$9B$22$7D
```

Server response

Code

Details

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "fcb0acc1-1b41-450b-bec8-bdb047f1f8e2",
      "title": "Моя кирпичная стена",
      "content": "Мой деревянный стул не выдержал..."
    },
    {
      "id": "027cf25-84fc-4e26-92d6-335ced596fe4",
      "title": "Моя кирпичные стены",
      "content": "Буду честен, я сделал..."
    },
    {
      "id": "a50ca562-14b5-446b-836a-5d56b5de3a8e",
      "title": "Кирпичные стены",
      "content": "А долго думать..."
    },
    {
      "id": "92b2b7b8-7337-416b-bef4-7c4249003f4f",
      "title": "Моя кирпичная дорожная стена",
      "content": "Поговорим об..."
    },
    {
      "id": "ca4df99e-b896-40f5-b0ff-73c8cea610bf",
      "title": "Моя кирпичные дорожные стены",
      "content": "Наконец я сделал очень хороший ремонт..."
    }
  ]
}
```

Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 833
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 18:58:17 GMT
etag: W/"341-V5ini0z5jcoM9Q0d1c1Co8dAjy"
keep-alive: timeout=5
x-powered-by: Express
```

Responses

Curl

```
curl -X 'GET' \
'http://localhost:5000/api/posts/public?filters$5Btitle$5D-$7B$2phraseSearch$22$3$A$22$D0$9$A$D0$B$8$D1$80$D0$B$F$D0$B$8$D1$87$D0$B$D0$B$0$D1$8F$2$0$D1$81$D1$82$D0$B$5$D0$B$D0$B$0$2$37D' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public?filters$5Btitle$5D-$7B$2phraseSearch$22$3$A$22$D0$9$A$D0$B$8$D1$80$D0$B$F$D0$B$8$D1$87$D0$B$D0$B$0$D1$8F$2$0$D1$81$D1$82$D0$B$5$D0$B$D0$B$0$2$37D
```

Server response

Code

Details

200

Response body

```
{
  "id": "fcb0acc1-1b41-4588-bee8-bd0b47f1f8e2",
  "title": "Моя кирпичная стена",
  "content": "Моя деревянный стул не выдержал..."
},
{
  "id": "027cfc25-84fc-4e26-92d6-335ced596fe4",
  "title": "Моя кирпичные стены",
  "content": "Буду честен, я сделал..."
},
{
  "id": "a50ca562-14b5-446b-836a-5d56b5dc3a8e",
  "title": "Кирпичные стены",
  "content": "Я долго думал..."
},
{
  "id": "92b2b7b8-7337-4160-bef4-7c4249003f4f",
  "title": "Моя кирпичная дорогая стена",
  "content": "Поговорим об..."
},
{
  "id": "ca4df99e-b896-4bf5-80ff-73c8cea610bf",
  "title": "Моя кирпичные дорогие стены",
  "content": "Недавно я сделал очень хороший ремонт..."
}
},
"message": "List of all posts"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 833
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 10:58:17 GMT
etag: W/"541-521a10e530d8qmodic1c0ddAjy"
keep-alive: timeout=5
x-powered-by: Express
```

Метод правильно вернул все посты. Это значит, что индекс в базе данных успешно был создан, а также все посты были успешно добавлены в данный индекс. Также это означает, что фразовый поиск правильно учитывает расстояние между словами и правильно учитывает разные формы слова и регистр.

Создание нового поста:

Теперь убедимся, что при создании нового поста – он будет добавлен не только в нашу базу данных, но и в базу данных Elasticsearch. Создадим такой пост:

```
{
  "title": "Почему футбол стал хуже",
  "access": "public",
  "content": "Сегодня обсудим одну очень наболевшую тему - современный футбол...",
  "tags": [
    "футбол"
  ]
}
```

Для этого выполним метод POST /api/posts/public и зададим этот пост:

The screenshot shows a REST client interface with the following sections:

- posts** (header)
- GET /api/posts/public** (method and endpoint)
- POST /api/posts/public** (method and endpoint)
- Parameters** (No parameters)
- Request body** (required, application/json)
- Examples:** [Modified value]
- Execute** (button)
- Clear** (button)
- Responses** (section header)
- Curl** (curl -X 'POST' \ 'http://localhost:5000/api/posts/public' \ -H 'accept: application/json' \ -H 'Content-Type: application/json' \ -d '{ "title": "Почему футбол стал хуже", "access": "public", "content": "Сегодня обсудим одну очень наболевшую тему - современный футбол...", "tags": ["футбол"] }')
- Request URL** (http://localhost:5000/api/posts/public)
- Server response** (201)
- Response body** (JSON response: { "status": 201, "data": { "id": "c856276c-26c5-42bc-a517-bbc0847f5736", "title": "Почему футбол стал хуже", "access": "public", "content": "Сегодня обсудим одну очень наболевшую тему - современный футбол...", "rating": 0, "tags": ["футбол"] }, "message": "Post successfully created" })
- Response headers** (access-control-allow-credentials: true, access-control-allow-origin: *, connection: keep-alive, content-length: 345, content-type: application/json; charset=utf-8, date: Wed, 07 Aug 2024 11:04:09 GMT, etag: W/"159-2ZAU5qR6rWIMBisvooyMNLKeo", keep-alive: timeout=5, location: /api/posts/c856276c-26c5-42bc-a517-bbc0847f5736, x-powered-by: Express)

Новый пост успешно создан. Теперь проверим – был ли он добавлен в базу данных Elasticsearch. Для этого выполним метод GET /api/posts/public, передав в качестве параметра поиска – title, а в качестве метода поиска – phraseSearch, указав в строке запроса 'футбол хуже':

Изменим этот пост с помощью метода PATCH `/api/posts/public/{postId}`, указав id данного поста:

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)	c856276c-26c5-42bc-a517-bbc0847f5736

Request body required

application/json

Examples:

[Modified value]

```
{
  "title": "Почему футбол стал лучше - нет"
}
```

Execute

Responses

Curl

```
curl -X 'PATCH' \
  'http://localhost:5000/api/posts/public/c856276c-26c5-42bc-a517-bbc0847f5736' \
  -H 'accept: application/json' \
  -H 'content-type: application/json' \
  -d '{
    "title": "Почему футбол стал лучше - нет"
  }'
```

Request URL

http://localhost:5000/api/posts/public/c856276c-26c5-42bc-a517-bbc0847f5736

Server response

Code

Details

200

Response body

```
{
  "status": 200,
  "data": {
    "id": "c856276c-26c5-42bc-a517-bbc0847f5736",
    "title": "Почему футбол стал лучше - нет",
    "access": "public",
    "content": "Сегодня обсудим одну очень популярную тему - современный футбол...",
    "rating": 0,
    "tags": [
      "футбол"
    ],
    "updatedAt": "2024-08-07T11:09:20.521Z"
  },
  "message": "Post successfully updated"
}
```

Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 395
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 11:09:20 GMT
etag: W/"18b-bbd0njs0VrgP50032g/tcr8sXB4"
keep-alive: timeout=5
x-powered-by: Express
```

Данный пост был успешно изменен. Убедимся в этом через метод GET `/api/posts/public`, передав в качестве параметра поиска – title, а в качестве метода поиска – phraseSearch, указав в строке запроса 'Футбол лучше'. Если данный пост будет найден, то это означает, что при изменении поста в локальной базе данных – он также изменяется в базе данных ElasticSearch:

posts

GET

/api/posts/public

Get all user posts

Parameters

Cancel

Name	Description
filters object (query)	Object that defines filters for searching among titles or contents <pre>{ "title": { "phraseSearch": "футбол лучше" } }</pre>

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/posts/public?filters%5Btitle%5D=%5B%7B%22phraseSearch%22%3A%22%20%1%84%20%1%83%20%1%82%20%1%81%20%1%80%20%1%82%20%1%83%20%1%84%20%1%85%22%7D' \
  -H 'accept: application/json'
```

Request URL

http://localhost:5000/api/posts/public?filters%5Btitle%5D=%5B%7B%22phraseSearch%22%3A%22%20%1%84%20%1%83%20%1%82%20%1%81%20%1%80%20%1%82%20%1%83%20%1%84%20%1%85%22%7D

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{ "status": 200, "data": [{ "id": "c856276c-26c5-42bc-a517-bbc0847f5736", "title": "Почему футбол стал лучше - нет", "content": "Сегодня обсудим одну очень интересную тему - современный футбол...", "message": "List of all posts" }] }</pre></div><div>Download</div></div> <div><div>Response headers</div><div><pre>access-control-allow-credentials: true access-control-allow-origin: * connection: keep-alive content-length: 297 content-type: application/json; charset=utf-8 date: Wed, 07 Aug 2024 11:12:11 GMT etag: W/"129-m8q4D+DKmc7R4afbx/s/2uK5R11" keep-alive: timeout=5 x-powered-by: Express</pre></div></div>

Как можно заметить – мы успешно получили данный пост. Значит он успешно был обновлен в базе данных Elasticsearch.

Удаление существующего поста:

Теперь убедимся, что после удаления поста из нашей базы данных – он также удаляется и в базе данных Elasticsearch. Для этого удалим данный пост:

```
{
  "id": "c856276c-26c5-42bc-a517-bbc0847f5736",
  "title": "Почему футбол стал лучше - нет",
  "content": "Сегодня обсудим одну очень наболевшую тему - современный футбол..."
}
```

Воспользуемся методом DELETE `/api/posts/public/{postId}`, передав id данного поста:

The screenshot displays a REST client interface with the following sections:

- posts** (header)
- Endpoints List:**
 - 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
 - DELETE `/api/posts/public/{postId}` Delete a post by id** (selected)
- Parameters:**
 - postId** (required, string, path): Post Id. Example: `550e8400-e29b-41d4-a716-446655440000`. Value: `c856276c-26c5-42bc-a517-bbc0847f5736`.
- Execute** button
- Responses:**
 - Curl:**

```
curl -X 'DELETE' \
'http://localhost:5000/api/posts/public/c856276c-26c5-42bc-a517-bbc0847f5736' \
-H 'accept: */*'
```
 - Request URL:** `http://localhost:5000/api/posts/public/c856276c-26c5-42bc-a517-bbc0847f5736`
 - Server response:**
 - Code:** 200
 - Response body:**

```
{
  "status": 200,
  "message": "Post successfully deleted"
}
```
 - Response headers:**

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 52
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 11:14:16 GMT
etag: W/"34-vdK6Q7dJ291dImyKruksAd678"
keep-alive: timeout=5
x-powered-by: Express
```

Пост был успешно удален.

Теперь нужно убедиться, что он был также удален из базы данных ElsticSearch. Для этого выполним метод GET /api/posts/public, передав в качестве параметра поиска – title, а в качестве метода поиска – phraseSearch, указав в строке запроса 'футбол лучше'. Если пост был успешно удален из базы данных ElsticSearch, то мы не должны получить его в результатах:

posts

GET

/api/posts/public

Get all user posts

Parameters

Cancel

Name	Description
filters	Object that defines filters for searching among titles or contents
object (query)	<pre>{ "title": { "phraseSearch": "другое лицо" } }</pre>

Execute

Clear

Responses

Curl

```
curl -X 'GET' \  
  'http://localhost:5000/api/posts/public?filters$5Btitle$5D=~$7B$22phraseSearch$22$3A$22$D1$84$D1$83$D1$82$D0$B1$D0$B$20$D0$B$D1$83$D1$87$D1$85$D0$B$K2$37D' \  
  -H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public?filters$5Btitle$5D=~$7B$22phraseSearch$22$3A$22$D1$84$D1$83$D1$82$D0$B1$D0$B$20$D0$B$D1$83$D1$87$D1$85$D0$B$K2$37D
```

Server response

CodeDetails

200

Response body

```
{  
  "status": 200,  
  "data": [],  
  "message": "List of all posts"  
}
```

Download

Response headers

```
access-control-allow-credentials: true  
access-control-allow-origin: *  
connection: keep-alive  
content-length: 54  
content-type: application/json; charset=utf-8  
date: Wed, 07 Aug 2024 11:15:44 GMT  
etag: W/"36-RJVLsudiB1VAJAKUK988xX0p/Qs"  
keep-alive: timeout=5  
x-powered-by: Express
```

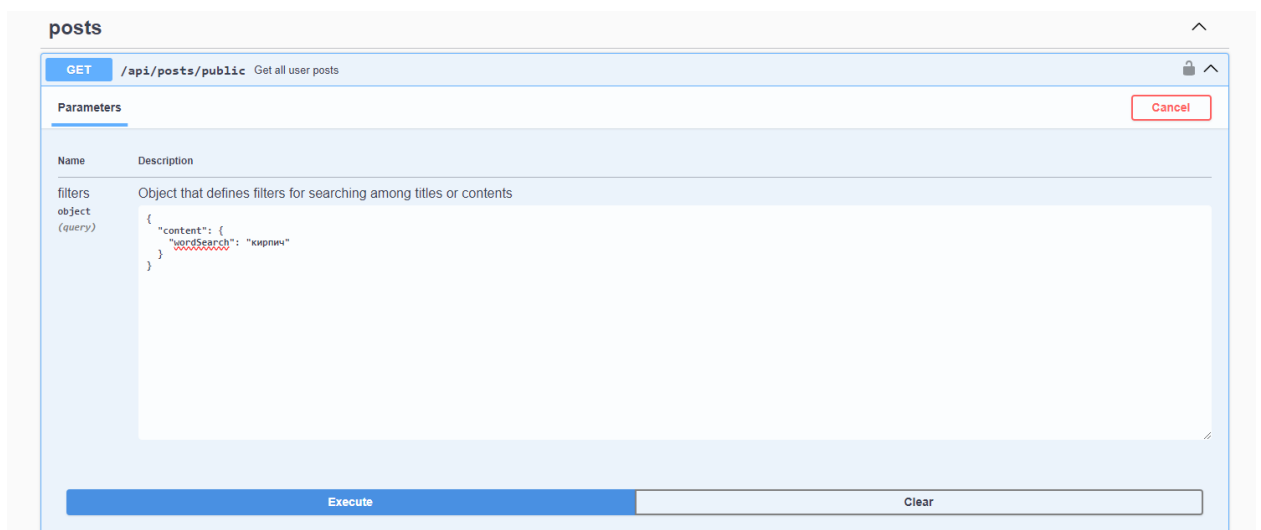
Мы не получили данный пост. Это значит, что действительно при удалении поста в нашей базе данных – он также удаляется и в базе данных ElasticSearch.

Пример полнотекстового поиска:

Сейчас мы постоянно выполняли фразовый поиск по колонке 'title' с текстом 'кирпичная стена'. Выполним сейчас полнотекстовый поиск по колонке 'content' с текстом 'кирпич'. Исходя из существующих постов пользователя, мы должны получить следующие посты:

```
{
  "id": "44150a1f-d02b-4542-9c8a-afc8b5856cd0",
  "title": "Ремонт дома",
  "access": "public",
  "content": "Мой кирпичный дом...",
  "rating": 0,
  "tags": null
},
{
  "id": "aaaa3f48-08cd-4c26-b980-f56654a69d3a",
  "title": "Очень интересная история",
  "access": "public",
  "content": "Я хочу сказать, что мой кирпич сегодня...",
  "rating": 0,
  "tags": null
}
```

Для этого выполним метод GET /api/posts/public, передав в качестве параметра поиска – content, а в качестве метода поиска – wordSearch, указав в строке запроса 'кирпич':



Curl

```
curl -X 'GET' \
'http://localhost:5000/api/posts/public?filters%5Bcontent%5D=%5C%Bx2wordSearch%2K%AkZ2KDQNBAAQDNBSND1S8QNDQBf3QDNBSND1N87Kz2K7D' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public?filters%5Bcontent%5D=%5C%Bx2wordSearch%2K%AkZ2KDQNBAAQDNBSND1S8QNDQBf3QDNBSND1N87Kz2K7D
```

Server response

CodeDetails

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "4415ba1f-d92b-4542-9c8a-a4cb5856cd0",
      "title": "Ремонт дома",
      "content": "Мой кирпичный дом..."
    },
    {
      "id": "aaaa3f48-08cd-4c26-b980-f56654a69d3a",
      "title": "Очень интересная история",
      "content": "Я хочу сказать, что мой кирпич сегодня..."
    }
  ],
  "message": "List of all posts"
}
```

Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
content-length: 367
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 11:19:44 GMT
etag: W/"16f-N7sPRZUKmiwMAZ/Ajo/3dlhuGJI"
x-powered-by: Express
```

Мы действительно получили правильные посты.

Проверка на сохранение документов в индексе при перезапуске Docker:

Убедимся, что данные действительно сохраняются, если перезапустить Docker. Для этого остановим работу Docker:

```
2926ad){cmHfNz6KQfOqpqvjAFwnXw}} is selected as the current health node.", "ecs.version": "1.2.0", "service.name": "ES_ECS", "event.dataset": "elasticsearch.server", "process.thread.name": "elasticsearch[572f822926ad][management][T#3]", "log.logger": "org.elasticsearch.health.node.selection.HealthNodeTaskExecutor", "elasticsearch.cluster.uuid": "vxWAmI3wRT2WL95vQceXWQ", "elasticsearch.node.id": "cmHfNz6KQfOqpqvjAFwnXw", "elasticsearch.node.name": "572f822926ad", "elasticsearch.cluster.name": "docker-cluster"}
elasticsearch_container | { "@timestamp": "2024-07-30T16:27:32.118Z", "log.level": "INFO", "current.health": "YELLOW", "message": "Cluster health status changed from [RED] to [YELLOW] (reason: [shards started [[post_idx][0]]]).", "previous.health": "RED", "reason": "shards started [[post_idx][0]]", "ecs.version": "1.2.0", "service.name": "ES_ECS", "event.dataset": "elasticsearch.server", "process.thread.name": "elasticsearch[572f822926ad][masterService#updateTask][T#1]", "log.logger": "org.elasticsearch.cluster.routing.allocation.AllocationService", "elasticsearch.cluster.uuid": "vxWAmI3wRT2WL95vQceXWQ", "elasticsearch.node.id": "cmHfNz6KQfOqpqvjAFwnXw", "elasticsearch.node.name": "572f822926ad", "elasticsearch.cluster.name": "docker-cluster"}
Gracefully stopping... (press Ctrl+C again to force)
[+] Stopping 1/1
✓ Container elasticsearch_container Stopped
^CСЗавершить выполнение пакетного файла [Y(да)/N(нет)]? y
PS C:\Users\Вордан\Desktop\Консорцеум Кодекс\Стажировка\daccord>
```

Перейдем по пути <http://localhost:9200>, чтобы убедиться, что порт 9200 перестал отвечать:

← ⓘ × localhost:9200

Страница недоступна

Страница не найдена

Если вы искали на ней что-то конкретное, спросите об этом Яндекс:

×

[Техническая информация](#)

Порт 9200 перестал отвечать. Теперь остановим также наше приложение, а после перезапустим Docker и приложение таким же способом, как в начале тестов:

```
time="2024-07-30T20:47:07+03:00" level=warning msg="C:\\Users\\Бордан\\Desktop\\Консорцеум Кодекс\\Стажировка\\dac
cord\\docker-compose.yaml: `version` is obsolete"
[+] Running 1/0
  ✓ Container elasticsearch_container Created 0.0s
Attaching to elasticsearch_container
elasticsearch_container | Jul 30, 2024 5:47:10 PM sun.util.locale.provider.LocaleProviderAdapter <clinit>
elasticsearch_container | WARNING: COMPAT locale provider will be removed in a future release
elasticsearch_container | {"@timestamp":"2024-07-30T17:47:11.470Z", "log.level": "INFO", "message":"Using [jdk] n
ative provider and native methods for [Linux]", "ecs.version": "1.2.0", "service.name": "ES_ECS", "event.dataset": "el
asticsearch.server", "process.thread.name": "main", "log.logger": "org.elasticsearch.nativeaccess.NativeAccess", "elast
icsearch.node.name": "572f822926ad", "elasticsearch.cluster.name": "docker-cluster"}
elasticsearch_container | {"@timestamp":"2024-07-30T17:47:11.847Z", "log.level": "INFO", "message":"Java vector i
QfOqpqvjAFwnXw", "elasticsearch.node.name": "572f822926ad", "elasticsearch.cluster.name": "docker-cluster"}
elasticsearch_container | {"@timestamp":"2024-07-30T17:47:23.268Z", "log.level": "INFO", "message":"Node [{572f82
2926ad}(cmHfNz6KQfOqpqvjAFwnXw)] is selected as the current health node.", "ecs.version": "1.2.0", "service.name": "
ES_ECS", "event.dataset": "elasticsearch.server", "process.thread.name": "elasticsearch[572f822926ad][management][T#3]
", "log.logger": "org.elasticsearch.health.node.selection.HealthNodeTaskExecutor", "elasticsearch.cluster.uuid": "vxWA
mI3wRT2WL95vQceXWQ", "elasticsearch.node.id": "cmHfNz6KQfOqpqvjAFwnXw", "elasticsearch.node.name": "572f822926ad", "ela
sticsearch.cluster.name": "docker-cluster"}
elasticsearch_container | {"@timestamp":"2024-07-30T17:47:23.298Z", "log.level": "INFO", "current.health": "YELLOW
W", "message": "Cluster health status changed from [RED] to [YELLOW] (reason: [shards started [[post_idx][0]])", "p
revious.health": "RED", "reason": "shards started [[post_idx][0]]", "ecs.version": "1.2.0", "service.name": "ES_ECS", "
event.dataset": "elasticsearch.server", "process.thread.name": "elasticsearch[572f822926ad][masterService#updateTask]
[T#1]", "log.logger": "org.elasticsearch.cluster.routing.allocation.AllocationService", "elasticsearch.cluster.uuid":
"vxWAmI3wRT2WL95vQceXWQ", "elasticsearch.node.id": "cmHfNz6KQfOqpqvjAFwnXw", "elasticsearch.node.name": "572f822926ad"
, "elasticsearch.cluster.name": "docker-cluster"}
View in Docker Desktop View Config Enable Watch
```

```
localhost:9200
Автоформатировать
{
  "name" : "572f822926ad",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "vxWAmI3wRT2WL95vQceXWQ",
  "version" : {
    "number" : "8.14.1",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "93a57a1a76f556d8aee6a90d1a95b06187501310",
    "build_date" : "2024-06-10T23:35:17.114581191Z",
    "build_snapshot" : false,
    "lucene_version" : "9.10.0",
    "minimum_wire_compatibility_version" : "7.17.0",
    "minimum_index_compatibility_version" : "7.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

```
{
  event: 'reverted',
  name: '20240719151942-add_text_tsv_column_to_post_table.js',
  durationSeconds: 0.022
}
Executing (default): SELECT "id", "name", "role", "email", "password", "isActivated", "refreshToken", "verifToken"
, "createdAt", "updatedAt", "rating" FROM "Users" AS "User" WHERE "User"."isActivated" = false;
Executing (default): SELECT "id", "name", "role", "email", "password", "isActivated", "refreshToken", "verifToken"
, "createdAt", "updatedAt", "rating" FROM "Users" AS "User" WHERE "User"."isActivated" = true AND "User"."verifTok
en" IS NOT NULL;
Server running on http://localhost:5000 - development
```

Все успешно перезапустилось.

После перезапуска, перейдем в Swagger и выполним опять метод GET /api/posts/public, передав в качестве параметра поиска – title, а в качестве метода поиска – phraseSearch, указав в строке запроса 'Кирпичная стена'. Если данные сохранились, то данный запрос должен вернуть нам все посты из предыдущих тестов по запросу 'Кирпичная стена':

posts

GET

/api/posts/public

Get all user posts


Parameters

Cancel

Name	Description
filters	Object that defines filters for searching among titles or contents
object (query)	<pre>{ "title": { "phraseSearch": "Кирпичная стена" } }</pre>

Execute

Clear

Responses	
<p>Curl</p> <pre>curl -X 'GET' \ 'http://localhost:5000/api/posts/public?filters:X5BtitleX5D~X7Bx22phraseSearchx22X3AK22X0X9A0X0X8BXND1X8X0ND0XBFX0X0X8ND1X87XND0XBDND0XND0ND1X8FX2XND1X81ND1X82XND0X85XND0XBDND0X8X2X7D' \ -H 'accept: application/json'</pre>	
<p>Request URL</p> <pre>http://localhost:5000/api/posts/public?filters:X5BtitleX5D~X7Bx22phraseSearchx22X3AK22X0X9A0X0X8BXND1X8X0ND0XBFX0X0X8ND1X87XND0XBDND0XND0ND1X8FX2XND1X81ND1X82XND0X85XND0XBDND0X8X2X7D</pre>	
<p>Server response</p> <p>Code Details</p>	
<p>200</p> <p>Response body</p> <pre>{ "status": 200, "data": [{ "id": "fcb0acc1-1b41-458b-bec8-b0b047f1f8e2", "title": "Моя кирпичная стена", "content": "Мой деревянный стул не выдержал..." }, { "id": "027cfc25-84fc-4e26-92d6-335ced596fe4", "title": "Моя кирпичные стены", "content": "Буду честен, я сделал..." }, { "id": "a50ca562-1405-446b-836a-5d56b5de3a8e", "title": "Кирпичные стены", "content": "А долго думал..." }, { "id": "92b2b7b8-7337-4160-bef4-7c4249003f4f", "title": "Моя кирпичная дорогая стена", "content": "Поговорим об..." }, { "id": "ca4df99e-b896-40f5-80ff-73c8cea610bf", "title": "Моя кирпичные дорожные стены", "content": "Наконец я сделал очень хороший ремонт..." }] }</pre>	 Download
<p>Response headers</p> <pre>access-control-allow-credentials: true access-control-allow-origin: * connection: keep-alive content-length: 833 content-type: application/json; charset=utf-8 date: Wed, 07 Aug 2024 10:58:17 GMT etag: W/"341-V5ini0z5joxMkQMd1c1Co8dAjy" keep-alive: timeout=5 x-powered-by: Express</pre>	

Responses

Curl

```
curl -X 'GET' \
'http://localhost:5000/api/posts/public?filters$5Btitle$5D-$7B$2phraseSearch$22K3ASZ2ND093AD0NB8ND1$80ND0NBFXD0NB8ND1$87AD0NBAD0NB0ND1$8F$20ND1$81ND1$82ND0NB5$D0NB0ND0NB0$22K7D' \
-H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public?filters$5Btitle$5D-$7B$2phraseSearch$22K3ASZ2ND093AD0NB8ND1$80ND0NBFXD0NB8ND1$87AD0NBAD0NB0ND1$8F$20ND1$81ND1$82ND0NB5$D0NB0ND0NB0$22K7D
```

Server response

Code Details

200

Response body

```
{
  "id": "fcb0acc1-1b41-4588-bee8-bd0b47f1f8e2",
  "title": "Моя кирпичная стена",
  "content": "Мой деревянный стул не выдержал..."
},
{
  "id": "027cfc25-84fc-4e26-92d6-335ced596fe4",
  "title": "Мои кирпичные стены",
  "content": "Буду честен, я сделал..."
},
{
  "id": "a50ca562-14b5-446b-836a-5d56b5dc3a8e",
  "title": "Кирпичные стены",
  "content": "Я долго думал..."
},
{
  "id": "92b2b7b8-7337-4160-be4-7c4249003f4f",
  "title": "Моя кирпичная дорогая стена",
  "content": "Поговорим об..."
},
{
  "id": "ca4df99e-b896-4bf5-80ff-73c8cea610bf",
  "title": "Мои кирпичные дорогие стены",
  "content": "Недавно я сделал очень хороший ремонт..."
}
},
"message": "List of all posts"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 833
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 10:58:17 GMT
etag: W/"5d110e530a89a0d1c0ddAjy"
keep-alive: timeout=5
x-powered-by: Express
```

Мы действительно при фразовом поиске получили все посты из предыдущих тестов. Это значит, что данные в индексе сохраняются, даже при перезапуске Docker.

Проверка на получение постов других пользователей:

Теперь проверим, что другой пользователь не может получать чужие посты при фразовом поиске. Для этого выйдем из системы:

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

Response body

```
{
  "status": 200,
  "message": "Logged out successfully"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
content-length: 50
content-type: application/json; charset=utf-8
date: Tue, 30 Jul 2024 17:57:30 GMT
etag: W/"32-uPVLuQmX0uQx9Td+XL8Q89CI6Q"
x-powered-by: Express
```

auth

GET /auth/payload Get a user payload

Parameters

No parameters

Execute Clear

Responses

Curl

```
curl -X 'GET' \
'http://localhost:5000/auth/payload' \
-H 'accept: application/json'
```

Request URL

http://localhost:5000/auth/payload

Server response

Code	Details
401 <small>Undocumented</small>	<p>Error: Unauthorized</p> <p>Response body</p> <p>Unauthorized - you are not signed in</p> <p>Response headers</p> <pre>access-control-allow-credentials: true access-control-allow-origin: * connection: keep-alive content-length: 36 content-type: text/html; charset=utf-8 date: Tue, 30 Jul 2024 17:58:04 GMT etag: W/"24-1pnbilyb3ip60XZvllipj14ybA" keep-alive: timeout=5 x-powered-by: Express</pre>

Мы успешно вышли из системы.

Теперь авторизируемся как другой пользователь:

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": "bgdghsH748LkKd???"
}
```

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": "bGdghsH748Lkdd?!"
  }'
```

Request URL

http://localhost:5000/auth/signin

Server response

Code Details

200

Response body

```
{
  "status": 200,
  "data": {
    "name": "ingaale",
    "email": "nozdryakovbogdan3112@mail.ru"
  },
  "message": "Authorization was successful"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 121
content-type: application/json; charset=utf-8
date: Tue, 30 Jul 2024 17:59:30 GMT
etag: W/"79-507Lfw4tzmPN1prA4WV/4bch80"
keep-alive: timeout=5
x-powered-by: Express
```

GET /auth/payload Get a user payload

Parameters

No parameters

Execute Clear

Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/auth/payload' \
  -H 'accept: application/json'
```

Request URL

http://localhost:5000/auth/payload

Server response

Code Details

200

Response body

```
{
  "status": 200,
  "data": {
    "name": "ingaale",
    "role": "user",
    "email": "nozdryakovbogdan3112@mail.ru"
  },
  "message": "User details"
}
```

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 119
content-type: application/json; charset=utf-8
date: Tue, 30 Jul 2024 18:00:47 GMT
etag: W/"77-IV+rDh8WJk10CoeinQasxHCBg"
keep-alive: timeout=5
x-powered-by: Express
```

Мы успешно авторизовались как другой пользователь.

Теперь выполним метод GET /api/posts/public без передачи параметров и посмотрим на посты данного пользователя:

posts

GET /api/posts/public Get all user posts

Parameters

filters

Object that defines filters for searching among titles or contents

object (query)

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

Code Details

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "61407480-6c70-474e-a338-d57c5e8ff7d7",
      "title": "Укрепленная кирпичная стена",
      "access": "public",
      "content": "Делим время ремонту...",
      "rating": 0,
      "tags": null
    },
    {
      "id": "c3eab93f-735b-4bd2-aedb-7c54bb01ca51",
      "title": "Дерево - лучшее решение",
      "access": "public",
      "content": "Новый деревянный стул меня очень...",
      "rating": 0,
      "tags": null
    },
    {
      "id": "0ccdb93b-4614-40cf-b4b7-e3f822ce5a9c",
      "title": "Мебель для дома",
      "access": "public",
      "content": "Новые деревянные стулья очень хороши...",
      "rating": 0,
      "tags": null
    }
  ]
}
```

Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 1598
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 11:23:06 GMT
etag: W/"7ce-MFG0208+bXL9b1XNN+KxBytdCo"
keep-alive: timeout=5
x-powered-by: Express
```

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

Code Details

200

Response body

```
{
  "status": 200,
  "data": [
    {
      "id": "61407480-6c70-474e-a338-d57c5e8ff7d7",
      "title": "Укрепленная кирпичная стена",
      "access": "public",
      "content": "Уделив время ремонту...",
      "rating": 0,
      "tags": null
    },
    {
      "id": "c3eab93f-735b-4bd2-aedb-7c54bb01ca51",
      "title": "Дерево - лучшее решение",
      "access": "public",
      "content": "Мой деревянный стул меня очень...",
      "rating": 0,
      "tags": null
    },
    {
      "id": "0ccdb93b-4614-40cf-b4b7-e3f822ce5a9c",
      "title": "Мебель для дома",
      "access": "public",
      "content": "Мои деревянные стулья очень хороши...",
      "rating": 0,
      "tags": null
    }
  ]
}
```



Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 1998
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 11:23:06 GMT
etag: W/"7ce-MFG0208r8X01X0M+kXBytdCo"
keep-alive: timeout=5
x-powered-by: Express
```

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

Code Details

200

Response body

```
{
  "id": "0d8b84f4-3f0a-4c3a-821e-7d3286cbaebc",
  "title": "Быстрое обновление",
  "access": "public",
  "content": "Мой деревянный стол не смог пройти испытание временем. Я решил все обновить, включая мои стулья...",
  "rating": 0,
  "tags": null
},
{
  "id": "197f2f63-4f39-44cc-a8cc-3fd3e5166631",
  "title": "Качественная мебель",
  "access": "public",
  "content": "Стул долго не сможет выдерживать такой вес, разве что деревянный сможет...",
  "rating": 0,
  "tags": null
},
{
  "id": "bd2bb6f1-59c9-44ae-95d7-660b1a65cfc7",
  "title": "Испанский футбол",
  "access": "public",
  "content": "Еще в детстве я влюбился в игру одного футболиста...",
  "rating": 0,
  "tags": null
},
{
  "id": "3be9cd5b-bb80-4294-8b9d-f957e1bd6726",
  "title": "IVE - мое имя",
  "access": "public",
  "tags": null
}
```



Download

Response headers

```
access-control-allow-credentials: true
access-control-allow-origin: *
connection: keep-alive
content-length: 1998
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 11:23:06 GMT
etag: W/"7ce-MFG0208r8X01X0M+kXBytdCo"
keep-alive: timeout=5
x-powered-by: Express
```

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

Code Details

200

Response body

```
{
  "id": "bd2bbbf1-59c9-44ae-95d7-660b1a65cfc7",
  "title": "Испанский футбол",
  "access": "public",
  "content": "Еще в детстве я влюбился в игру одного футболиста...",
  "rating": 0,
  "tags": null
},
{
  "id": "39e9cd5b-bb80-4294-8b9d-f957e1bd6726",
  "title": "IVE - лучше",
  "access": "public",
  "content": "Недавно я познакомился с одной кроп группой - IVE. Я влюбился в их девочек...",
  "rating": 0,
  "tags": null
},
{
  "id": "9f954444-c95a-4984-91d6-d702c71411f4",
  "title": "Лучшее средство от эрозии...",
  "access": "public",
  "content": "Конечно, использовать деревянный...",
  "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: 1998
content-type: application/json; charset=utf-8
date: Wed, 07 Aug 2024 11:23:06 GMT
etag: W/"7ce-MFG0Z08rbXLI9b1X0W+KxBytdCo"
keep-alive: timeout=5
x-powered-by: Express
```

То есть данный пользователь имеет следующие посты:

```
{
  "id": "61407480-6c70-474e-a338-d57c5e8ff7d7",
  "title": "Укрепленная кирпичная стена",
  "access": "public",
  "content": "Уделим время ремонту...",
  "rating": 0,
  "tags": null
},
{
  "id": "c3eab93f-735b-4bd2-aedb-7c54bb01ca51",
  "title": "Дерево - лучшее решение",
  "access": "public",
  "content": "Мой деревянный стул меня очень...",
  "rating": 0,
  "tags": null
},
```

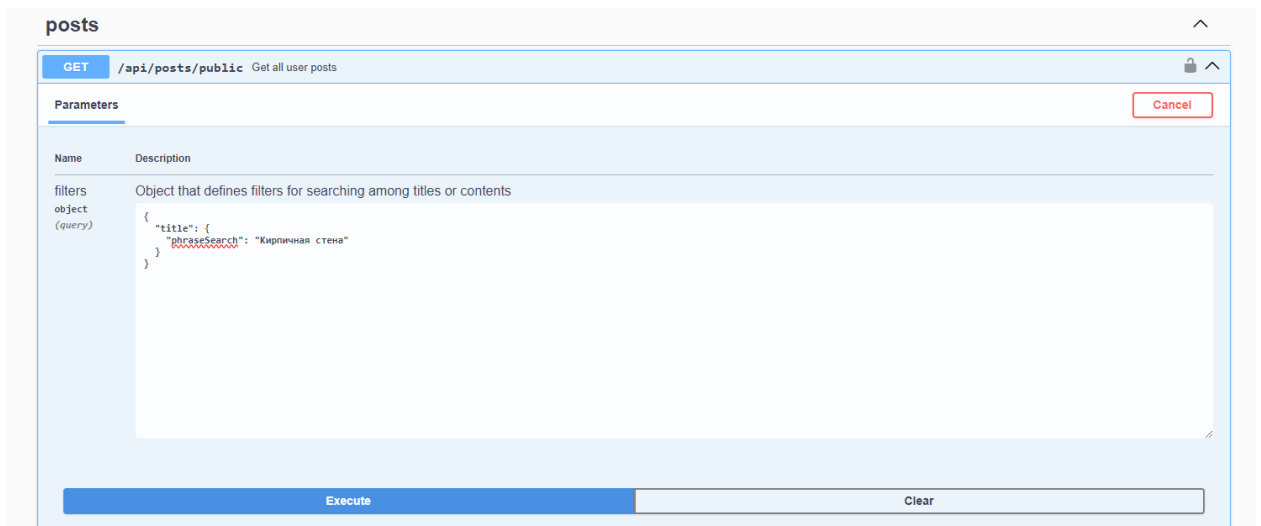
```
{
  "id": "0ccdb93b-4614-40cf-b4b7-e3f822ce5a9c",
  "title": "Мебель для дома",
  "access": "public",
  "content": "Мои деревянные стулья очень хороши...",
  "rating": 0,
  "tags": null
},
{
  "id": "0d8b84f4-3f0a-4c3a-821e-7d3286cbaebc",
  "title": "Быстрое обновление",
  "access": "public",
  "content": "Мой деревянный стол не смог пройти испытание временем. Я решил все обновить, включая мои стулья...",
  "rating": 0,
  "tags": null
},
{
  "id": "197f2f63-4f39-44cc-a8cc-3fd3e5166631",
  "title": "Качественная мебель",
  "access": "public",
  "content": "Стул долго не сможет выдерживать такой вес, разве что деревянный сможет...",
  "rating": 0,
  "tags": null
},
{
  "id": "bd2bbbf1-59c9-44ae-95d7-660b1a65cfc7",
  "title": "Испанский футбол",
  "access": "public",
  "content": "Еще в детстве я влюбился в игру одного футболиста...",
  "rating": 0,
  "tags": null
}
```

```

},
{
  "id": "39e9cd5b-bb80-4294-8b9d-f957e1bd6726",
  "title": "IVE - лучшие",
  "access": "public",
  "content": "Недавно я познакомился с одной крор группой - IVE. Я влюбился в их девочек...",
  "rating": 0,
  "tags": null
},
{
  "id": "9f954444-c95a-4984-91d6-d702c71411f4",
  "title": "Лучшее средство от эрозии...",
  "access": "public",
  "content": "Конечно, использовать деревянный...",
  "rating": 0,
  "tags": null
}
}

```

Теперь выполним метод GET /api/posts/public, передав в качестве параметра поиска – title, а в качестве метода поиска – phraseSearch, указав в строке запроса 'Кирпичная стена'. Данный пользователь имеет только один пост, где есть указанная строка в title. Данный метод не должен возвращать посты предыдущего пользователя, где в title также есть указанная строка:



Responses

Curl

```
curl -X 'GET' \
  'http://localhost:5000/api/posts/public?filters$5Btitle$5D-$7B$2phraseSearch$2K3A$22ND0$9AND0$B8ND1$80ND0$BFND0$B8ND1$87ND0$B7ND0$B0ND1$8F$20ND1$81ND1$82ND0$BF5ND0$B7AD0$B0$22$7D' \
  -H 'accept: application/json'
```

Request URL

```
http://localhost:5000/api/posts/public?filters$5Btitle$5D-$7B$2phraseSearch$2K3A$22ND0$9AND0$B8ND1$80ND0$BFND0$B8ND1$87ND0$B7ND0$B0ND1$8F$20ND1$81ND1$82ND0$BF5ND0$B7AD0$B0$22$7D
```

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{ "status": 200, "data": [{ "id": "61407480-6c70-474e-a338-d57c5e8ff7d7", "title": "Укрепленная кирпичная стена", "content": "Укрепим опору прочности..." }], "message": "List of all posts" }</pre></div><div><div>Download</div></div></div> <div><div>Response headers</div><div><pre>access-control-allow-credentials: true access-control-allow-origin: * connection: keep-alive content-length: 216 content-type: application/json; charset=utf-8 date: Wed, 07 Aug 2024 11:28:48 GMT etag: W/"db-6LkQ3PBWqL4s3qTV/7SA-HzPU" keep-alive: timeout=5 x-powered-by: Express</pre></div></div>

Пользователь получил именно свой пост и не получил посты предыдущего пользователя. Значит все работает правильно – фразовый поиск позволяет искать пользователям только свои посты.

Еще один пример полнотекстового поиска:

Выполним метод полнотекстовый поиск для текущего пользователя по полю content, передав в запрос 'Деревянный стул'. Исходя из существующих постов пользователя, мы должны получить следующие посты:

```
{
  "id": "c3eab93f-735b-4bd2-aedb-7c54bb01ca51",
  "title": "Дерево - лучшее решение",
  "access": "public",
  "content": "Мой деревянный стул меня очень...",
  "rating": 0,
  "tags": null
},
{
  "id": "0ccdb93b-4614-40cf-b4b7-e3f822ce5a9c",
  "title": "Мебель для дома",
  "access": "public",
  "content": "Мои деревянные стулья очень хороши...",
  "rating": 0,
```

```
    "tags": null
  },
  {
    "id": "0d8b84f4-3f0a-4c3a-821e-7d3286cbaebc",
    "title": "Быстрое обновление",
    "access": "public",
    "content": "Мой деревянный стол не смог пройти испытание временем. Я решил все обновить, включая мои стулья...",
    "rating": 0,
    "tags": null
  },
  {
    "id": "197f2f63-4f39-44cc-a8cc-3fd3e5166631",
    "title": "Качественная мебель",
    "access": "public",
    "content": "Стул долго не сможет выдерживать такой вес, разве что деревянный сможет...",
    "rating": 0,
    "tags": null
  },
}
```

Для этого выполним метод GET /api/posts/public для текущего пользователя, передав в качестве параметра поиска – content, а в качестве метода поиска – wordSearch, указав в строке запроса 'Деревянный стул':

