

Backend-разработка на Go. Модуль 2

# Elasticsearch

---



# На этом уроке

1. Узнаем, как применять Elasticsearch в качестве движка полнотекстового поиска в структурированных данных на Go.
2. Научимся синхронизировать данные между другими хранилищами и Elasticsearch.
3. Изучим язык запросов Elastic — Query DSL, а также возможности делать SQL-запросы к Elastic.

## Оглавление

### [Теория урока](#)

[Запуск ElasticSearch](#)

[Набор данных для примера](#)

[Скачиваем данные из STAPI и сохраняем их в Elasticsearch из Go](#)

[Читаем данные Elasticsearch в Go](#)

[Поиск документов в Elasticsearch через Elastic Query DSL](#)

[SQL-запросы к Elasticsearch](#)

### [Практическое задание](#)

### [Глоссарий](#)

### [Дополнительные материалы](#)

# Теория урока

## Запуск Elasticsearch

Elasticsearch — распространённое хранилище данных для любых типов информации, которое поддерживает распределённую структуру (горизонтальное масштабирование) для обеспечения скорости и отказоустойчивости, а также индексирует многие типы контента, что делает его одним из самых популярных инструментов для полнотекстового поиска. Это хранилище использует простое REST-API.

Для Go есть официальная [библиотека](#), которую мы используем дальше.

Сегодня мы узнаем, как создать простое приложение, которое позволяет добавлять данные и выполнять поиск в Elasticsearch через Go.

Чтобы потренироваться с использованием Elasticsearch на локальном компьютере, запустим его через docker. Для этого создадим скрипт для запуска **run-elastic.sh**:

```
#!/bin/bash
docker rm -f elasticsearch
docker run -d --name elasticsearch -p 9200:9200 -e discovery.type=single-node \
  -v elasticsearch:/usr/share/elasticsearch/data \
  docker.elastic.co/elasticsearch/elasticsearch:7
docker ps
```

Запустим этот скрипт, сделаем его исполняемым и откроем в своём браузере ссылку <http://localhost:9200>, чтобы увидеть информацию о сервере в формате json.

## Набор данных для примера

Сайт [Star Trek API](#) содержит большой объём данных о вселенной Star Trek. Введём URL-адрес <http://stapi.co/api/v1/rest/spacecraft/search?pageNumber=0&pageSize=100&pretty> в браузер, чтобы увидеть объект JSON, содержащий информацию о странице и о космических кораблях. Всего насчитывается более 1 200 космических аппаратов.

Воспользуемся данными о космических кораблях в качестве набора данных для нашего приложения.

## Скачиваем данные из STAPI и сохраняем их в Elasticsearch из Go

Напишем простую программу Go, которая подключается к Elasticsearch и распечатывает информацию о сервере. Создадим файл с именем **simple.go**, содержащий:

```

package main

import (
    "github.com/elastic/go-elasticsearch/v7"
    "log"
)

func main() {
    es, err := elasticsearch.NewDefaultClient()
    if err != nil {
        log.Fatalf("Error creating the client: %s", err)
    }
    log.Println(elasticsearch.Version)

    res, err := es.Info()
    if err != nil {
        log.Fatalf("Error getting response: %s", err)
    }
    defer res.Body.Close()
    log.Println(res)
}

```

И запустим его:

```
go run simple.go
```

Мы увидим информацию о сервере, аналогичную той, что мы видели раньше в браузере.

Теперь создадим пользовательский интерфейс в виде простого консольного приложения, управляемого текстовым меню.

Создадим файл с именем elastic.go, содержащий код Go:

```

import (
    "bufio"
    "fmt"
    "os"
    "github.com/elastic/go-elasticsearch/v7"
)

var es, _ = elasticsearch.NewDefaultClient()

func main() {
    reader := bufio.NewScanner(os.Stdin)
    for {
        fmt.Println("0) Exit")
        fmt.Println("1) Load spacecraft")
        fmt.Println("2) Get spacecraft")
    }
}

```

```

    option := ReadText(reader, "Enter option")
    if option == "0" {
        Exit()
    } else if option == "1" {
        LoadData()
    } else {
        fmt.Println("Invalid option")
    }
}
}

```

Код ещё не готов к запуску. Нам надо написать функцию LoadData (). В этой функции мы прочитаем все данные о космических кораблях с сайта STAPI. Сайт позволяет одновременно читать не более 100 записей, поэтому данные содержатся на 13 страницах. Это означает, что нам потребуется прочитать каждую страницу по очереди.

Создадим файл с именем loaddata.go, содержащий такой код:

```

import (
    "context"
    "encoding/json"
    "io/ioutil"
    "net/http"
    "strconv"
    "strings"

    "github.com/elastic/go-elasticsearch/esapi"
)

func LoadData() {
    var spacecrafts []map[string]interface{}
    pageNumber := 0
    for {
        response, _ :=
http.Get("http://stapi.co/api/v1/rest/spacecraft/search?pageSize=100&pageNumber=
" + strconv.Itoa(pageNumber))
        body, _ := ioutil.ReadAll(response.Body)
        defer response.Body.Close()
        var result map[string]interface{}
        json.Unmarshal(body, &result)

        page := result["page"].(map[string]interface{})
        totalPages := int(page["totalPages"].(float64))

        crafts := result["spacecrafts"].([]interface{})

        for _, craftInterface := range crafts {
            craft := craftInterface.(map[string]interface{})
            spacecrafts = append(spacecrafts, craft)
        }
    }
}

```

```

        pageNumber++
        if pageNumber >= totalPages {
            break
        }
    }

    for _, data := range spacecrafts {
        uid, _ := data["uid"].(string)
        jsonString, _ := json.Marshal(data)
        request := esapi.IndexRequest{Index: "stsc", DocumentID: uid, Body:
strings.NewReader(string(jsonString))}
        request.Do(context.Background(), es)
    }
    print(len(spacecrafts), " spacecraft read\n")
}

```

Код создаёт переменную с именем `spacecrafts`, содержащую пустой слайс-мап. Каждая мапа имеет строковые ключи. Значения могут быть любого типа.

Начинаем перебирать страницы начиная с нулевой. У нас будет бесконечный цикл для выборки страниц данных, а завершится он, когда прочитается последняя страница. Затем через Go http API извлекаем страницу данных из STAPI, указывая её номер. Тело ответа — это объект JSON, который парсится в мапу `result`.

Эта мапа после парсинга станет содержать две записи: мапу `page` с информацией о текущей странице и слайс информации о космических аппаратах `spacecrafts`. В `page` нас интересует только общее количество страниц, поэтому извлекаем его в переменную с именем `totalPages`. Затем код выполняет итерацию по списку космических аппаратов и использует приведение типа для представления каждой записи в виде мапы. Далее запись добавляется к слайсу космических аппаратов `spacecrafts`. Код увеличивает номер страницы и, если это последняя страница, завершает бесконечный цикл через `break`.

Теперь у нас есть слайс, содержащий все космические корабли, каждая запись которого представляет собой мапу, куда входят данные о космическом корабле. Пришло время сохранить данные в Elasticsearch.

Данные вставляются в Elasticsearch путём создания переменной типа `esapi.IndexRequest`.

Элементы данных в Elasticsearch называются документами. Elasticsearch хранит документы в коллекциях, называемых индексами. Каждому документу присваивается уникальный идентификатор в индексе, поэтому мы используем `uid` космического корабля в качестве уникального идентификатора индекса. Для основной части документа сериализуем данные космического корабля в JSON.

Операция вставки фактически выполняется путём вызова функции `Do()`, с передачей ей контекста `Go` и клиента `Elastic` (глобальная переменная `es`). Запускаем программу и выбираем пункт меню для загрузки данных.

Теперь проверим содержание данных в индексе `stsc`, набрав в веб-браузере адрес [http://localhost:9200/stsc/\\_search](http://localhost:9200/stsc/_search). Перед нами появятся некоторые данные из индекса.

## Читаем данные Elasticsearch в Go

Загрузка документов в Elasticsearch оказалась сложной из-за обязательного преобразования данных через промежуточные мапы и слайсы. Получение и поиск документов будет намного проще. Изменения, которые мы внесём, требуют нового импорта, поэтому начнём с обновления нашей секции импорта:

```
import (
    "bufio"
    "bytes"
    "context"
    "encoding/json"
    "fmt"
    "os"

    "github.com/elastic/go-elasticsearch/esapi"
    "github.com/elastic/go-elasticsearch/v8"
)
```

Elasticsearch возвращает документы в виде объекта JSON, содержащего метаданные и содержимое документа. Добавим в `Elastic.go` функцию `Print()`, которая распечатывает информацию о космическом корабле в более удобочитаемой форме.

```
func Print(spacecraft map[string]interface{}) {
    name := spacecraft["name"]
    status := ""
    if spacecraft["status"] != nil {
        status = "- " + spacecraft["status"].(string)
    }
    registry := ""
    if spacecraft["registry"] != nil {
        registry = "- " + spacecraft["registry"].(string)
    }
    class := ""
    if spacecraft["spacecraftClass"] != nil {
        class = "- " +
            spacecraft["spacecraftClass"].(map[string]interface{})["name"].(string)
    }
}
```

```
fmt.Println(name, registry, class, status)
}
```

Функция учитывает тот факт, что некоторые поля могут отсутствовать. Документы запрашиваются путём указания индекса и идентификатора документа. Добавим ещё один пункт меню в Elastic.go, чтобы получать космический корабль по запросу. Меню будет вызывать функцию Get():

```
fmt.Println("0) Exit")
fmt.Println("1) Load spacecraft")
fmt.Println("2) Get spacecraft")
fmt.Println("3) Search spacecraft by key and value")
fmt.Println("4) Search spacecraft by key and prefix")
option := ReadText(reader, "Enter option")
if option == "0" {
    Exit()
} else if option == "1" {
    LoadData()
} else if option == "2" {
    Get(reader)
} else if option == "3" {
    Search(reader, "match")
} else if option == "4" {
    Search(reader, "prefix")
} else {
    fmt.Println("Invalid option")
}
```

```
func Get(reader *bufio.Scanner) {
    id := ReadText(reader, "Enter spacecraft ID")
    request := esapi.GetRequest{Index: "stsc", DocumentID: id}
    response, _ := request.Do(context.Background(), es)
    var results map[string]interface{}
    json.NewDecoder(response.Body).Decode(&results)
    Print(results["_source"].(map[string]interface{}))
}
```

Документ возвращается в виде объекта JSON, который декодируется в `map[string]interface{}`. Сам документ — в ключе `_source`.

## Поиск документов в Elasticsearch через Elastic Query DSL

Elasticsearch поддерживает несколько различных типов поиска. В каждой операции поиска требуется указать тип запроса и набор ключей-значений для отбора данных. Результат — список совпадений, каждому из которых присвоена оценка, показывающая, насколько хорошим было совпадение.



Значения поиска всегда входят в нижний регистр. Например, поиск по названию `uss` будет находить все космические корабли со словом `uss` в названии без учёта регистра, включая `USS`. Поиск по префиксу соответствует любому слову, которое начинается с указанной строки.

Выше мы уже добавили пункты меню для поиска. Теперь реализуем их.

```
func Search(reader *bufio.Scanner, querytype string) {
    key := ReadText(reader, "Enter key")
    value := ReadText(reader, "Enter value")
    var buffer bytes.Buffer
    query := map[string]interface{}{
        "query": map[string]interface{}{
            querytype: map[string]interface{}{
                key: value,
            },
        },
    }
    json.NewEncoder(&buffer).Encode(query)
    response, _ := es.Search(es.Search.WithIndex("stsc"),
    es.Search.WithBody(&buffer))
    var result map[string]interface{}
    json.NewDecoder(response.Body).Decode(&result)
    for _, hit := range
result["hits"].(map[string]interface{}).["hits"].([]interface{}) {
        craft :=
hit.(map[string]interface{}).["_source"].(map[string]interface{})
        Print(craft)
    }
}
```

После получения ключа и значения от пользователя функция создаёт структуру данных из типа запроса, ключа и значения. Затем он кодируется как объект JSON. Функция `es.Search()` вызывается с именем индекса и запросом в теле. Она возвращает список совпадений. Проходимся по нему и печатаем каждый документ, находящийся в поле `_source`.

Запустите программу и поищите совпадения по префиксу или полные.

Например:

1. Для опции меню 3 — «Поиск космических аппаратов по ключу и значению»:
  - Enter key: name Enter value: enterprise.
2. Для опции меню 4 — «Поиск по ключу и префиксу космического корабля»:
  - Enter key: registry Enter value: ncc;
  - или Enter key: name Enter value: iks.

# SQL-запросы к Elasticsearch

Кроме собственного диалекта языка запросов Elastic Query DSL, используются также псевдозапросы SQL. Поддерживаются два API для работы с SQL-запросами.

Первое API выполняет запросы:

```
POST /_sql?format=txt
{
  "query": "SELECT * FROM library WHERE release_date < '2000-01-01'"
}
```

Результат:

author	name	page_count	release_date
Dan Simmons	Hyperion	482	1989-05-26T00:00:00.000Z
Frank Herbert	Dune	604	1965-06-01T00:00:00.000Z

Второе API выполняет преобразование SQL-запросов в запросы Elastic Query DSL:

```
POST /_sql/translate
{
  "query": "SELECT * FROM library ORDER BY page_count DESC",
  "fetch_size": 10
}
```

Результат:

```
{
  "size": 10,
  "_source": false,
  "fields": [
    {
      "field": "author"
    },
    {
      "field": "name"
    },
    {
      "field": "page_count"
    },
    {
      "field": "release_date",
      "format": "strict_date_optional_time_nanos"
    }
  ]
}
```

```
],
"sort": [
  {
    "page_count": {
      "order": "desc",
      "missing": "_first",
      "unmapped_type": "short"
    }
  }
]
}
```

API применяется для трансформации, так как SQL API работает довольно медленно. На каждый запрос Elastic производит трансформацию в свой внутренний язык и лишь затем исполняет. Этот промежуточный этап можно исключить, выполнив предварительную трансформацию.

## Практическое задание

1. Создайте сервис, который по REST API получает запросы на добавление карточек продуктов и запросы на их поиск. Условия:
  - a. Каждый продукт, как в интернет-магазине, имеет название и несколько любых других реквизитов.
  - b. Запрос на поиск:
    - выполняется таким образом, чтобы поиск осуществлялся по всем возможным полям продукта;
    - ранжируется так, чтобы поиск по наименованию был приоритетнее, то есть с более высокой оценкой, среди поиска по другим полям.

## Глоссарий

1. **Elasticsearch** — распространённое хранилище данных для любых типов информации, которое поддерживает распределённую структуру (горизонтальное масштабирование) для обеспечения скорости и отказоустойчивости, а также индексирует многие типы контента, что делает его одним из самых популярных инструментов для полнотекстового поиска.

## Дополнительные материалы

1. Статья [Multi-match query](#).
2. Статья [Search API](#).
3. Статья [SQL](#).

4. Статья [SQL CLI](#).