Update from 4
Administrator создал 1 месяц назад

0e75f26a

README.md 16,5 КБ

Day 03 - Go Boot camp

Tasty Discoveries

Contents

Chapter I

General rules

- Your programs should not quit unexpectedly (giving an error on a valid input). If this happens, your project will be considered non functional and will receive a 0 during the evaluation.
- We encourage you to create test programs for your project even though this work won't have to be submitted and won't be graded. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded.
- If your code is using external dependencies, it should use Go Modules for managing them

Chapter II

Rules of the day

- You should only turn in `*.go` files and (in case of external dependencies) `go.mod` + `go.sum`
- Your code for this task should be buildable with just `go build`
- All inputs ('page'/'lat'/'long') should be thouroughly validated and never cause HTTP 500 (only HTTP 400/401 is acceptable, with a meaningful error message, as explained in EX02)

Chapter III

Intro

People tend to love some recommending apps. It helps to avoid thinking too much about what to buy, where to go and what to eat.

Also, pretty much everyone has a phone with a geolocation. How often did you try finding some restaurants in your area for dinner?

Let's think a bit about how these services work and build one of our own, really simple one, shall we?

Chapter IV

Exercise 00: Loading Data

There are lots and lots of various databases on the market. But, because we're trying to provide the ability to search for things, let's use Elasticsearch. All examples provided have been tested on version 7.9.2.

Elasticsearch is a full text search engine built on top of Lucene. It provides an HTTP API that we will be using in this task.

Our provided dataset of restaurants (taken from an Open Data portal) consists of more than 13 thousands of restaurants in the area of Moscow, Russia (you can put together another similar dataset for any other location you want). Every entry has:

- ID
- Name
- Address
- Phone
- Longitude
- Latitude

Before uploading all entries into the database, let's create an index and a mapping (explicitly specifying data types). Without it Elasticsearch will try to guess field types based on documents provided, and sometimes it won't recognize geopoints.

Here are a couple links to help you get started on things:

- https://www.elastic.co/guide/en/elasticsearch/reference/7.9/indices-create-index.html
- https://www.elastic.co/guide/en/elasticsearch/reference/7.9/geo-point.html

Start the database by running `~$ /path/to/elasticsearch/dir/bin/elasticsearch` and let's experiment around.

For simplicity, let's use "places" as a name for an index and "place" as a name for an entry. You can create an index using cURL like this:

```
~$ curl -XPUT "http://localhost:9200/places"
```

but in this task you should use Go Elasticsearch bindings to do the same thing. Next thing you have to do is to provide type mappings for our data. With cURL it will look like this:

```
~$ curl -XPUT http://localhost:9200/places/place/_mapping?include_type_name=true -H "Content-Type: application/json" -d @"schema.json"
```

where `schema.json` looks like this:

```
{
  "properties": {
    "name": {
        "type":  "text"
    },
    "address": {
        "type":  "text"
    },
    "phone": {
        "type":  "text"
    },
    "location": {
      "type": "geo_point"
    }
  }
}
```

Once again, provided cURL commands are just a reference for self-testing, this action should be performed by the Go program you write.

Now you have a dataset to upload. You should use Bulk API to perform that. All existing Elasticsearch bindings provide wrappers for it, for example, here is a good example for an official client (keep in mind that you'll need to use client v7 for ES version 7.9, not v8). There are also a couple of third-party clients, choose whichever you prefer.

To check yourself, you may use cURL. So,

```
~$ curl -s -XGET "http://localhost:9200/places"
```

should give you something like this:

```
{
  "places": {
    "aliases": {},
    "mappings": {
      "properties": {
        "address": {
          "type": "text"
        },
        "id": {
          "type": "long"
        },
        "location": {
          "type": "geo_point"
        },
        "name": {
          "type": "text"
        },
        "phone": {
          "type": "text"
        }
      }
    },
    "settings": {
      "index": {
        "creation_date": "1601810777906",
        "number_of_shards": "1",
        "number_of_replicas": "1",
        "uuid": "4JKa9fgISd6-N130rpNYtQ",
        "version": {
          "created": "7090299"
        },
        "provided_name": "places"
      }
    }
  }
}
```

and querying entry by its ID will look like this:

```
~$ curl -s -XGET "http://localhost:9200/places/place/1"
```

```
{
  "_index": "places",
  "_type": "place",
  "_id": "1",
  "_version": 1,
  "_seq_no": 0,
  "_primary_term": 1,
  "found": true,
  "_source": {
    "id": 1,
    "name": "SMETANA",
    "address": "gorod Moskva, ulitsa Egora Abakumova, dom 9",
    "phone": "(499) 183-14-10",
    "location": {
      "lat": 55.879001531303366,
      "lon": 37.71456500043604
    }
  }
}
```

Please note, that the entry with ID=1 may be different from the one in dataset if you decided to use goroutines to speed up the process (that's not a requirement in this task though).

Chapter V

Exercise 01: Simplest Interface

Now let's create an HTML UI for our database. Not much, we just need to render a page with a list of names, addresses and phones so user could see it in a browser.

You should abstract your database behind an interface. To just return the list of entries and be able to paginate through them, this interface is enough:

```
type Store interface {
    // returns a list of items, a total number of hits and (or) an error in case of one
    GetPlaces(limit int, offset int) ([]types.Place, int, error)
}
```

There should be to Elasticsearch-related imports in `main` package, as all database-related stuff should rest in `db` package inside your project, and you should only use this interface above to interact with it.

Your HTTP application should run on port 8888, responding with a list of restaurants and providing a simple pagination over it. So. when querying "http://127.0.0.1:8888/?page=2" (mind the 'page' GET param) you should be getting a page like this:

```
<!doctype html>
<html>
```

```
<head>
    <meta charset="utf-8">
    <title>Places</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>

<body>
<h5>Total: 13649</h5>
<ul>
    <li>
        <div>Sushi Wok</div>
        <div>gorod Moskva, prospekt Andropova, dom 30</div>
        <div>(499) 754-44-44</div>
    </li>
    <li>
        <div>Ryba i mjaso na ugljah</div>
        <div>gorod Moskva, prospekt Andropova, dom 35A</div>
        <div>(499) 612-82-69</div>
    </li>
    <li>
        <div>Hleb nasuschnyj</div>
        <div>gorod Moskva, ulitsa Arbat, dom 6/2</div>
        <div>(495) 984-91-82</div>
    </li>
    <li>
        <div>TAJJ MAHAL</div>
        <div>gorod Moskva, ulitsa Arbat, dom 6/2</div>
        <div>(495) 107-91-06</div>
    </li>
    <li>
        <div>Balalaechnaja</div>
        <div>gorod Moskva, ulitsa Arbat, dom 23, stroenie 1</div>
        <div>(905) 752-88-62</div>
    </li>
    <li>
        <div>IL Pizzaiolo</div>
        <div>gorod Moskva, ulitsa Arbat, dom 31</div>
        <div>(495) 933-28-34</div>
    </li>
    <li>
        <div>Bufet pri Astrahanskih banjah</div>
        <div>gorod Moskva, Astrahanskij pereulok, dom 5/9</div>
        <div>(495) 344-11-68</div>
    </li>
    <li>
        <div>MU-MU</div>
        <div>gorod Moskva, Baumanskaja ulitsa, dom 35/1</div>
        <div>(499) 261-33-58</div>
    </li>
    <li>
        <div>Bek tu Blek</div>
        <div>gorod Moskva, Tatarskaja ulitsa, dom 14</div>
        <div>(495) 916-90-55</div>
    </li>
    <li>
        <div>Glav Pirog</div>
        <div>gorod Moskva, Begovaja ulitsa, dom 17, korpus 1</div>
        <div>(926) 554-54-08</div>
    </li>
</ul>
<a href="/?page=1">Previous</a>
<a href="/?page=3">Next</a>
<a href="/?page=1364">Last</a>
</body>
</html>
```

A "Previous" link should disappear on page 1 and "Next" link should disappear on last page.

IMPORTANT NOTE: You may notice that by default Elasticsearch doesn't allow you to deal with pagination for more than 10000 entries. There are two ways to othercome this - either use a Scroll API (refer to the same link on pagination above) or just raise the limit in index settings specifically for this task. The latter is acceptable for this task, but is not the recommended way to do it in production. The query that will help you to set it is below:

```
~$ curl -XPUT -H "Content-Type: application/json" "http://localhost:9200/places/_settings" -d '
{
  "index" : {
    "max_result_window" : 20000
  }
}'
```

Also, in case 'page' param is specified with a wrong value (outside [0..last_page] or not numeric) your page should return HTTP 400 error and plain text with an error description:

```
Invalid 'page' value: 'foo'
```

Chapter VI

Exercise 02: Proper API

In modern world most applications prefer APIs over just plain HTML. So, in this exercise all you have to do is implement another handler which responds with `Content-Type: application/json` and JSON version of the same thing as in Ex01 (example for http://127.0.0.1:8888/api/places?page=3):

```
{
  "name": "Places",
  "total": 13649,
  "places": [
    {
      "id": 65,
      "name": "AZERBAJDZhAN",
      "address": "gorod Moskva, ulitsa Dem'jana Bednogo, dom 4",
      "phone": "(495) 946-34-30",
      "location": {
        "lat": 55.769830485601204,
        "lon": 37.486914061171504
      }
    },
    {
      "id": 69,
      "name": "Vojazh",
      "address": "gorod Moskva, Beskudnikovskij bul'var, dom 57, korpus 1",
      "phone": "(499) 485-20-00",
      "location": {
        "lat": 55.872553383512496,
        "lon": 37.538326789741
      }
    },
    {
      "id": 70,
      "name": "GBOU Shkola № 1411 (267)",
      "address": "gorod Moskva, ulitsa Bestuzhevyh, dom 23",
      "phone": "(499) 404-15-09",
      "location": {
        "lat": 55.87213179130298,
        "lon": 37.609625999999984
      }
    },
    {
      "id": 71,
      "name": "Zhigulevskoe",
      "address": "gorod Moskva, Bibirevskaja ulitsa, dom 7, korpus 1",
      "phone": "(964) 565-61-28",
      "location": {
        "lat": 55.88024342230735,
        "lon": 37.59308635976602
      }
    },
    {
      "id": 75,
      "name": "Hinkal'naja",
      "address": "gorod Moskva, ulitsa Marshala Birjuzova, dom 16",
      "phone": "(499) 728-47-01",
      "location": {
        "lat": 55.79476126986192,
        "lon": 37.491709793339744
      }
    },
    {
      "id": 76,
      "name": "ShAURMA ZhI",
      "address": "gorod Moskva, ulitsa Marshala Birjuzova, dom 19",
      "phone": "(903) 018-74-64",
      "location": {
        "lat": 55.794378830665885,
        "lon": 37.49112002224252
      }
    },
    {
      "id": 80,
      "name": "Bufet Shkola № 554",
      "address": "gorod Moskva, Bolotnikovskaja ulitsa, dom 47, korpus 1",
      "phone": "(929) 623-03-21",
      "location": {
        "lat": 55.66186417434049,
        "lon": 37.58323602169326
      }
    },
    {
      "id": 83,
      "name": "Kafe",
      "address": "gorod Moskva, 1-j Botkinskij proezd, dom 2/6",
      "phone": "(495) 945-22-34",
      "location": {
        "lat": 55.781141341601696,
        "lon": 37.55643137063551
      }
    },
    {
      "id": 84,
      "name": "STARYJ BATUM'",
      "address": "gorod Moskva, ulitsa Akademika Bochvara, dom 7, korpus 1",
      "phone": "(495) 942-44-85",
      "location": {
        "lat": 55.8060307318284,
        "lon": 37.461669109923506
      }
    },
    {
      "id": 89,
      "name": "Cheburechnaja SSSR",
      "address": "gorod Moskva, Bol'shaja Bronnaja ulitsa, dom 27/4",
      "phone": "(495) 694-54-76",
      "location": {
        "lat": 55.764134959774346,
        "lon": 37.60256453956346
      }
    }
  ],
  "prev_page": 2,
  "next_page": 4,
  "last_page": 1364
}
```

Also, in case 'page' param is specified with a wrong value (outside [0..last_page] or not numeric) your API should respond with a corresponding HTTP 400 error and similar JSON:

```
{
    "error": "Invalid 'page' value: 'foo'"
}
```

Chapter VII

Exercise 03: Closest Restaurants

Now let's implement our main piece of functionality - searching for *three* closest restaurants! In order to do that, you'll have to configure sorting for your query:

```
"sort": [
    {
      "_geo_distance": {
        "location": {
          "lat": 55.674,
          "lon": 37.666
        },
        "order": "asc",
        "unit": "km",
        "mode": "min",
        "distance_type": "arc",
        "ignore_unmapped": true
      }
    }
]
```

where "lat" and "lon" are your current coordinates. So, for an URL http://127.0.0.1:8888/api/recommend?lat=55.674&lon=37.666 your application should return JSON like this:

```
{
  "name": "Recommendation",
  "places": [
    {
      "id": 30,
      "name": "Ryba i mjaso na ugljah",
      "address": "gorod Moskva, prospekt Andropova, dom 35A",
      "phone": "(499) 612-82-69",
      "location": {
        "lat": 55.67396575768212,
        "lon": 37.66626689310591
      }
    },
    {
      "id": 3348,
      "name": "Pizzamento",
      "address": "gorod Moskva, prospekt Andropova, dom 37",
      "phone": "(499) 612-33-88",
      "location": {
        "lat": 55.673075576456,
        "lon": 37.664533747576
      }
    },
    {
      "id": 3347,
      "name": "KOFEJNJa «KAPUChINOFF»",
      "address": "gorod Moskva, prospekt Andropova, dom 37",
      "phone": "(499) 612-33-88",
      "location": {
        "lat": 55.672865251005106,
        "lon": 37.6645689561318
      }
    }
  ]
}
```

Chapter VIII

Exercise 04: JWT

So, the last (but not least) thing that we have to do is to provide some simple form of authentication. Currently the one of the most popular ways of implementing that for an API is by using JWT. Luckily, Go has a pretty good set of tooling to deal with it.

First, you have to implement an API endpoint http://127.0.0.1:8888/api/get_token which sole purpose will be to generate a token and return it, like this (this is an example, your token will likely be different):

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhZG1pbiI6dHJ1ZSwiZXhwIjoxNjAxOTc1ODI5LCJuYW1lIjoiTmlrb2xheSJ9.FqsRe0t9YhvEC3hK1pCWu
}
```

Don't forget to set header 'Content-Type: application/json'.

Second, you have to protect your `/api/recommend` endpoint with a JWT middleware, that will check the validity of this token.

So by default when querying this API from the browser it should now fail with an HTTP 401 error, but work when `Authorization: Bearer <token>` header is specified by the client (you may check this using cURL or Postman).

This is a simplest way to provide authentication, no need to go deeper in details for now.