Zadanie rekrutacyjne (GO)

Należy napisać prosty serwer HTTP przechowujący w pamięci lub bazie danych (w wersji rozszerzonej) zapisane urle oraz pobrane dane, udostępniający operacje CRUD (Create/Read/Update/Delete) na tych danych jako API RESTowe.

Założenia: - Serwer powinien nasłuchiwać na porcie `8080`. - Serwer zwraca odpowiedzi w formacie JSON. - Klucz `id` powinien być integerem. - Maksymalny rozmiar payloadu `POST` to `4KB`. - Worker powinien pobierać cyklicznie dane z `url` co `interval` sekund (nie mniej niż 5s). - Worker powinien mieć ustawiony timeout 5s na pobranie `url` - Unittesty i/lub testy integracyjne

Poniżej opis endpointów HTTP, które powinny być udostępniane przez serwer oraz ich przykładowe wywołanie z użyciem programu `curl`.

Tworzenie nowego obiektu - POST /api/urls

Zadaniem tego endpointu jest wpisywanie nowej wartości obiektu pod oczekiwany klucz.

Poprawne wywołanie i odpowiedź serwera w formacie JSON.

```
$ curl -si 127.0.0.1:8080/api/urls -X POST -d '{"url":"https://httpbin.org/range/15","interval":60}'
HTTP/1.1 201 Created
{"id": 1}
```

Niepoprawne wywołanie (nieprawidłowy JSON) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls -X POST -d 'niepoprawny json'
HTTP/1.1 400 Bad Request
```

Niepoprawne wywołanie (za duży obiekt) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls -X POST -d 'ponad 4 KB danych...'
HTTP/1.1 413 Request Entity Too Large
```

Aktualizacja obiektu - PATCH /api/urls/{id}

Zadaniem tego endpointu jest aktualizacja wartości dla istniejącego obiektu.

Poprawne wywołanie i odpowiedź serwera w formacie JSON.

```
$ curl -si 127.0.0.1:8080/api/urls/1 -X PATCH -d '{"url":"https://httpbin.org/range/15","interval":120}'
HTTP/1.1 200 OK
{"id": 1}
```

Niepoprawne wywołanie (nieistniejący klucz) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls/99 -X PATCH -d '{"url":"https://httpbin.org/range/15","interval":120}'
HTTP/1.1 404 OK
```

Niepoprawne wywołanie (za duży obiekt) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls/1 -X PATCH -d 'ponad 1 MB danych...'
HTTP/1.1 413 Request Entity Too Large
```

Niepoprawne wywołanie (nieprawidłowy klucz) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls/abc -X POST -d '...'
HTTP/1.1 400 Bad Request
```

Niepoprawne wywołanie (nieprawidłowy JSON) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls/1 -X POST -d 'niepoprawny json'
HTTP/1.1 400 Bad Request
```

Usuwanie obiektu - DELETE /api/urls/{id}

Zadaniem tego endpointu jest usunięcie wskazanego obiektu.

Poprawne wywołanie i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls/12 -X DELETE
HTTP/1.1 204 No Content
```

Niepoprawne wywołanie (nieistniejący klucz) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls/12 -X DELETE
HTTP/1.1 404 Not Found
```

Niepoprawne wywołanie (nieprawidłowy klucz) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls/abc -X DELETE
HTTP/1.1 400 Bad Request
```

Listowanie zapisanych obiektów - GET /api/urls

Zadaniem tego endpointu jest wylistowanie istniejących obiektów.

Poprawne wywołanie i odpowiedź serwera w formacie JSON.

```
$ curl -s 127.0.0.1:8080/api/urls
[
  {"id": 1, "url": "https://httpbin.org/range/15", "interval": 60},
  {"id": 2, "url": "https://httpbin.org/delay/10", "interval": 120}
]
```

Pobieranie historii pobrań - GET /api/urls/{id}/history

Zadaniem tego endpointu jest pobranie pełnej historii pobrań dla wskazanego obiektu.

Poprawne wywołanie i odpowiedź serwera w formacie JSON.

```
$ curl -si 127.0.0.1:8080/api/urls/1/history
HTTP/1.1 200 OK
[
  {
    "response": "abcdefghijklmno",
    "duration": 0.571,
    "created_at": 1559034638.315,
  },
  {
    "response": null,
    "duration": 5.000,
    "created_at": 1559034938.623,
  }
]
```

Gdzie `created_at` oznacza czas rozpoczęcia pobierania jako unix timestamp (float), `duration` to czas pobierania w sekundach (float).

Niepoprawne wywołanie (nieprawidłowy klucz) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls/abc
HTTP/1.1 400 Bad Request
```

Niepoprawne wywołanie (nieistniejący klucz) i odpowiedź serwera.

```
$ curl -si 127.0.0.1:8080/api/urls/99
HTTP/1.1 404 Not Found
```

Worker pobierający dane z urli

Ma za zadanie pobierać w tle dane z `url` z zadanym `interval` dla każdego obiektu. W przypadku błędu do pola `response` powinien zostać zapisany null. Do pola `response` zapisujemy cały response jaki zwróci `url`.

Na przykład:

```
curl -si httpbin.org/range/50
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwx
```

Czyli pod response zapisujemy "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwx"

Przygotowanie obrazu dockerowego z aplikacją

Rozwiązanie powinno zawierać plik `Dockerfile` umożliwiający utworzenie obrazu dockerowego z aplikacją:

```
$ docker build -t app .
```

Uruchomienie:

```
$ docker run -it --rm -p 8080:8080 app
```

Ewentualne dodatkowe parametry dla konfiguracji można przekazywać jako zmienne środowiskowe:

```
$ docker run -it --rm -e ZMIENNA1=100 -e ZMIENNA2=200 app
```

Część rozszerzona (baza danych)

Przygotowanie serwera i workera jako osobnych komponentów z wykorzystaniem dowolnej bazy danych.

Zbudowanie obrazu serwera i workera:

```
$ docker build -t app .
```

Uruchomienie serwera (+/- zmienne konfiguracyjne jako zmienne :

```
$ docker run -it --rm -p 8080:8080 app --server
```

Uruchomienie workera:

```
$ docker run -it --rm app --worker
```

Baza danych również zamknięta w obrazie dockerowym, np: - mysql - mongo - redis - lub dowolna inna

Uruchomienie wszystkich komponentów za pomocą docker-compose

```
$ docker-compose up
```

Część rozszerzona (otwarte pytania)

Załóżmy, że serwis został wdrożony produkcyjnie i jest dostępny publicznie z internetu.

- Jakie problemy mogą się pojawić w trakcie działania serwisu?
- Ile zapytań i danych serwis jest w stanie obsłużyć?
- W jaki sposób można się przygotować na większy ruch?
- W jaki sposób sprawdzić, że wszystko działa? Co monitorować?
- W jaki sposób zabezpieczyć serwis przed złośliwymi zapytaniami?

Wskazówki

Proponujemy i polecamy: - `github.com/go-chi/chi` jako router HTTP - `github.com/stretchr/testify` oraz `github.com/vektra/mockery` do testowania - `github.com/golangci/golangci-lint` jako linter - `https://hub.docker.com/_/golang` jako obraz bazowy do zbudowania obrazu dockerowego - `go modules` dla zarządzania zależnościami

Kod źródłowy powinien zostać umieszczony w serwisie `github.com` jako prywatne repozytorium. Dostarczymy listę użytkowników, którzy będą mogli mieć dostęp do rozwiązania.

W razie jakichkolwiek wątpliwości/pytań zapraszamy do kontaktu - chętnie pomożemy.