Fetch Take-Home Exercise — Site Reliability Engineering

Overview

Implement a program to check the health of a set of HTTP endpoints. A basic overview is given in this section. Additional details are provided in the Prompt section.

Read an input argument to a file path with a list of HTTP endpoints in YAML format. Test the health of the endpoints every 15 seconds. Keep track of the availability percentage of the HTTP domain names being monitored by the program. Log the cumulative availability percentage for each domain to the console after the completion of each 15-second test cycle.

Prompt

Sample input file

```
- headers:
    user-agent: fetch-synthetic-monitor
  method: GET
  name: fetch.com index page
  url: https://fetch.com/
- headers:
    user-agent: fetch-synthetic-monitor
  method: GET
  name: fetch.com careers page
  url: https://fetch.com/careers
- body: '{"foo":"bar"}'
  headers:
    content-type: application/json
    user-agent: fetch-synthetic-monitor
  method: POST
  name: fetch.com some post endpoint
  url: https://fetch.com/some/post/endpoint
- name: www.fetchrewards.com index page
  url: https://www.fetchrewards.com/
```

Parsing the program input

The program must accept a single, required input argument to a configuration file path. A well-formed configuration file is a YAML list. Each entry in the YAML list follows a consistent schema, and contains enough information for the program to send a well-formed HTTP request. You may assume that the contents of a configuration file given to your program are valid (you do not need to validate the schema of the configuration file).

An example of a valid YAML configuration file is included in the Sample input file section. While you may use this particular file for testing and validation purposes, your program must accept an arbitrary file path as its input.

Each HTTP endpoint element in the YAML list has the following schema:

- `name` (**string, required**) — A free-text name to describe the HTTP endpoint.
- `url` (**string, required**) — The URL of the HTTP endpoint.
  - You may assume that the URL is always a valid HTTP or HTTPS address.
- `method` (**string, optional**) — The HTTP method of the endpoint.
  - If this field is present, you may assume it's a valid HTTP method (e.g. `GET`, `POST`, etc.).
  - If this field is omitted, the default is `GET`.
- `headers` (**dictionary, optional**) — The HTTP headers to include in the request.
  - If this field is present, you may assume that the keys and values of this dictionary are strings that are valid HTTP header names and values.
  - If this field is omitted, no headers need to be added to or modified in the HTTP request.
- `body` (**string, optional**) — The HTTP body to include in the request.
  - If this field is present, you should assume it's a valid JSON-encoded string. You do not need to account for non-JSON request bodies.
  - If this field is omitted, no body is sent in the request.

Running the health checks

After parsing the YAML input configuration file, the program should send an HTTP request to each endpoint every 15 seconds. Each time an HTTP request is executed by the program, determine if the outcome is `UP` or `DOWN`:

- `UP` — The HTTP response code is 2xx (any 200–299 response code) and the response latency is less than 500 ms.
- `DOWN` — The endpoint is not `UP`.

Keep testing the endpoints every 15 seconds until the user manually exits the program.

Logging the results

Each time the program finishes testing all the endpoints in the configuration file, log the **availability percentage** of each **URL domain** over the lifetime of the program to the console.

Availability percentage is defined as:

```
100 * (number of HTTP requests that had an outcome of UP / number of HTTP requests)
```

The example YAML file in the Sample input file section contains two URL domains: `fetch.com` and `www.fetchrewards.com`. Note that the number of domains present in a configuration file can differ from the number of endpoints present in the file (two domains vs. four endpoints in this particular file).

Here's the complete expected output to the console of a sample execution of the program:

```
fetch.com has 33% availability percentage
www.fetchrewards.com has 100% availability percentage
fetch.com has 67% availability percentage
www.fetchrewards.com has 50% availability percentage
```

Here is the internal program state that led to the above output:

Program execution begins at time = 0 seconds. Program reads and parses the sample configuration file given as its input argument.

Test cycle #1 begins at time = 0 seconds:

- *Endpoint with name* `fetch.com index page` *has HTTP response code 200 and response latency 100 ms => UP*
- *Endpoint with name* `fetch.com careers page` *has HTTP response code 200 and response latency 600 ms => DOWN (response latency is not less than 500 ms)*
- *Endpoint with name* `fetch.com some post endpoint` *has HTTP response code 500 and response latency 50 ms => DOWN (response code is not in range 200–299)*
- *Endpoint with name* `www.fetchrewards.com index page` *has HTTP response code 200 and response latency 100 ms => UP*

Test cycle #1 ends. The program logs to the console:

```
fetch.com has 33% availability percentage
www.fetchrewards.com has 100% availability percentage
```

This is the expected output because 1/3 of `fetch.com` *health checks are* `UP` *and 1/1* `www.fetchrewards.com` *health checks are* `UP`*.*

Test cycle #2 begins at time = 15 seconds:

- *Endpoint with name* `fetch.com index page` *has HTTP response code 200 and response latency 100 ms => UP*
- *Endpoint with name* `fetch.com careers page` *has HTTP response code 200 and response latency 300 ms => UP*
- *Endpoint with name* `fetch.com some post endpoint` *has HTTP response code 201 and response latency 50 ms => UP*
- *Endpoint with name* `www.fetchrewards.com index page` *has HTTP response code 200 and response latency 900 ms => DOWN (response latency is not less than 500 ms)*

Test cycle #2 ends. The program logs to the console:

```
fetch.com has 67% availability percentage
www.fetchrewards.com has 50% availability percentage
```

This is the expected output because 4/6 of `fetch.com` *health checks are* `UP` *and 1/2* `www.fetchrewards.com` *health checks are* `UP` *since the program began running.*

User presses `CTRL+C` *and the program exits.*

Note that when logging the availability percentage for each domain, the program should round floating-point availability percentages to the nearest whole percentage.

FAQ

How will my solution be graded?

An engineer will review the code you submit. At a minimum, they must be able to run the code and it must produce the expected results. While your solution does not need to be fully production-ready, you are being evaluated, so put your best foot forward.

What programming language should I use?

You may use any language. Choose something that allows you to showcase what you know.

Can I use third-party open source software?

Yes, you may use publicly available third-party open source software to assist with your solution. If you include third-party components, you should ensure that your solution's instructions allow the exercise reviewer to successfully build and run your code.

How do I submit my solution?

We strongly recommend uploading your solution to a publicly accessible git-based repository system (e.g. Github, Gitlab, Bitbucket). This will allow the fastest and easiest way for the reviewer to assess your solution.

Do I need to provide instructions to the reviewer in my repository?

You should assume that the reviewer has no familiarity with your programming language of choice. You should therefore provide instructions in a `README` or similar format on how to build and run your solution. Keep in mind that the exercise reviewer may have a different operating system or software installed on their local machine, so make minimal assumptions in your instructions.

Should I use a durable data store (e.g. disk, SQL/NoSQL databases) for keeping track of availability percentages?

Definitely not. For the purposes of this exercise, it is fine to use a suitable data structure in your application's memory to keep track of and log the expected program output over time as each testing cycle completes. We do not expect your solution to persist data durably across multiple executions of your program.

How long do I have to complete the exercise?

There is no time limit for the exercise. We have designed the exercise so that it should take a few hours. But please take as much time as you need to put your best solution forward.

I have a question about the problem statement.

Use your best judgment to design and implement a program that meets the requirements of the prompt. Anything not explicitly stated is left to your discretion.