SRE Challenge

Hello! This README outlines two exercises and should be accompanied by a zip file containing the initial code. Each exercise is representative of a real problem we have run across in our time as SREs.

Solutions to each exercise will be evaluated on the following criteria:

- Completeness
- Correctness
- Tests
- Legibility
- Aesthetics

If you are unable to resolve the issues there could be good reasons (it could be our fault) so we would still like to talk through your problem solving.

Pre-requisites

The project provided was built with, and tested against, Docker *19.03.2* and Docker Compose *1.24.1*. To avoid complications, we strongly encourage you to use an equivalent installation. Each application running in a container is isolated from any local version concerns.

If you absolutely prefer to experiment with the web application on your workstation, please note that it was confirmed against Ruby *2.4.2*. We recommend rbenv and Bundler *1.16.0*.

Submission

To submit your work please create a *private* repo at github.com or gitlab.com with your changes and invite the following usernames to your project so they can review the changes:

- @bobeckert
- @AaronEast
- @jonnymccullagh
- @magdallena.trajkovska (gitlab) @magdallenatr (github)
- @jgjorgji-slice

- @kaircosta (gitlab) @hvithval (github)

Exercises

The below exercises involve troubleshooting and augmenting an application stack composed of:

1. a Sinatra HTTP API
2. being served by Puma
3. all fronted by nginx

To ensure your focus on the intended challenges, please note that the following components *do not need to be modified* to complete these exercises:

- any existing Ruby source code
- the ruby and nginx base images

Part 1: Finish Whats Been Started

The `sre` web service is an older backend service. To date, it had a "traditional" Ruby lifecycle: Bundler for local development, gem for packaging, and Puppet for deployment to EC2 hosts. In an effort to improve our developer workflow and modernize our operational model, the team decided to update the project to use Docker.

An industrious engineer recently added Docker Compose to tackle local development. Being new to Docker, however, they're wrestling a bit with the wiring between containers. While a `docker-compose up` will run both nginx and the `sre` web service, the former is unable to proxy requests to the latter.

Troubleshoot and remediate the issue with container networking. In the end, you should be able to successfully hit nginx as a proxy to the web service:

```
❯ curl localhost/sres
{"sres":["bob","linda","tina","gene","louise"]}⏎
```

Part 2: And Now for Something Completely Different

With the stack now functioning in Docker, there is an early opportunity to make it more cloud-native. The `sre` web service exposes a `/health` endpoint which had previously been used by load balancers and host agents to gather insights:

```
❯ curl localhost/health
{"health":true,"metrics":{"requestLatency":0.6009920113999009,"dbLatency":0.2865695923858
```

The team would like to use a container sidecar in this new architecture to maintain parity.

Please build -- and test the functionality of! -- a sidecar container. A specification follows. Once complete, the container should be added to Docker Compose to be brought up with nginx and the `sre` web service.

`sre` sidecar specification

The intent of this sidecar is to facilitate the health checking and insights gathering functions of the load balancers and metrics agents that preceded it. In order to do so, it will need to:

1. Poll `/health` on the web service every 10 seconds
2. Output the average, minimum, and maximum of each metric every minute to stdout
3. Exit with a non-zero exit code if `/health` does not return 200