Developer's notes

David Rubin

david.rubin95@gmail.com

February 6, 2023

I. Repository

The source code is hosted on Github and is available at `https://github.com/rubinda/logtopus`. The README document contains all necessary instructions to run and test the application. This document provides an outline of the development process and only short usage instructions. As per the email instructions, the repository is private and accessible by the user mentioned in the email. Please contact me in case of issues.

II. Design choices

All the functionality is developed using Go using the standard libraries with a few exceptions - JSON web tokens, environment variable parsing and server lifecycle management use external libraries. The core HTTP server functionality uses the standard `net/http` library.

For the storage mechanism I've chosen InfluxDB. The requirement to capture invariant data made me look for a solution in the NoSQL space. Because the instructions specified that it would be a write-intensive service, I've considered solutions which focus on such workloads. My two main candidates were Cassandra and InfluxDB. Since Cassandra recommends designing your data model based on your desired queries and the desired solution should query on custom runtime defined fields, I've chosen InfluxDB. It is a time-series database, which advertises high performance [1]. I've proposed a very basic structure for captured events, with the intention of it being extensible or modifiable in the future. To keep this document short, more details are available in the repository readme (under the `/events` endpoint) or in the source code (under `pkg/influxdb/models.go`).

For deploying the application I've included docker configuration files. It does require docker to be installed in the host system, but I find the portability, simplicity of deployment and isolation from the target host as great benefits.

III. Usage instructions

After cloning the repository, one can use docker compose from the root project directory to start 2 containers:

```
docker compose up
```

One can also include the `-d` flag, so that docker runs in detached mode. Cleanup can be done with `docker compose down` and removing the images (`docker image rm <image_name>`). This will start 2 containers: one running the application code

[1] https://www.influxdata.com/blog/influxdb-vs-cassandra-time-series/

listening on port 5000, and another one for running InfluxDB.

The running application can be invoked using `cURL`. The base endpoint is:

```
https://localhost:5000/api/v1/
```

More details are found in the repository readme: `https://github.com/rubinda/logtopus#usage`.

IV. Future steps

My initial idea was to structure the application with the use of interfaces. This would make the dependencies (like storage mechanism or authentication provider) trivial and they could easily be replaced without reaching into other parts of the project. But I've chosen to keep it simple for the sake of demonstration purposes.

InfluxDB was chosen based on the few examples provided in the instructions. I assumed there would be attributes for each event and large or highly complex schemas would not be present. Since it is a time-series database, each query must be done with a time range (with a *quick and easy* solution to query since the beginning of time). This also makes it required that each event possesses a timestamp.

Furthermore, the query functionality was secondary in terms of designing the service, since it is deemed to be a write intensive application. InfluxDB offers an official Go client library, but at the time of development, it didn't support parametrized queries for custom InfluxDB deployments (only their cloud solution supports it). Hence I've developed a very crude query builder, which only offers querying by exact field value.

InfluxDB offers several storage data types including user defined data objects but since little is known about the data model, I've only kept the very basic types including some limitations. One can currently store and retrieve integers, floats, strings, booleans, lists (which are converted into lists of strings regardless of given type) and objects, which are stored as JSON strings.

Endpoint naming is somewhat unusual because I've chosen to query via POST requests and use JSON documents as query descriptors instead of URL parameters and GET requests. The idea was for the client to provide data types in queries since InfluxDB's query language (Flux) requires the field value to be the same type as the stored value (e.g. once a field value is given as a float, one can not store another record with the same field name as a string or query with string values).

Many functionalities use hardcoded strings for the simplicity and ease of development. I've tried to outline a more robust solution using constants in a few places, which I think is appropriate once a project starts growing and has more collaborators (reduces risk of typos etc.)

Endpoints are encrypted using TLS (with the downside of self signed certificates during development) and two require authentication via JSON web tokens. For proof of concept I find it a straightforward solution to implement but based on the use cases a better approach might be available. What comes to mind are automated services which produce large amounts of data in short intervals and would require to also query for tokens. One could use a common secret, or a public-private key-pair method to reduce the amount of unnecessary requests.

When contemplating implementation design choices I've followed the Occam's razor principle and kept the most simple pattern that I found appropriate. I find such an approach more appropriate when a proof of concept is needed and the specifications leave open questions. Given more information about the desired solution or the ability to ask questions might have steered me into a somewhat different approach.