DBD exam report

Henning W.

May 31, 2021

1 Introduction

Choice of application: Some kind of social media platform thing. Due to large time constraints, some suboptimal solutions will have be taken. I usually would start off the project by making CI/CD with a server. These factors have been discarded entirely. This means, that there won't be any online capabilities in this project. Instead the entire application will be containerized, in an attempt at making the local solution easier to run. Another suboptimal solution is, that the database containers and the containers which utilize them have been split into two separate container files. You'd usually want both of them to share the same docker-compose file. The problem lies with migration strategies, and what I like calling "A chicken or egg" complication. Both data processing and my backend are reliant on the database. Further more, my backend is reliant on the data-processing. Because of this, the containers will need to be run in chunks. The standard solution to this is somewhat suboptimal in itself. It involves files which lets other processes run first before executing itself.

Since the project requires a larger amount of data, the project has been created a bit backwards. I've looked at some datasets, and then slowly formulated data models from them.

I'm using a Rust backend, since it's a solo project, and I already have a setup for it. It's effective and stable. It's a role I don't trust Python all that much with.

The Python container for data processing is only run once. It's purpose is to populate the databases with some dummy data. The data is acquired from several sources, and translated via. the Python library Pandas, and then subsequently formatted and populated into Mongo. From there, the data will once again be formatted and populated into Meilisearch. The reason why the Python container isn't populating the Postgres database as well, is because it'd mean, that both the Rust backend and the Python data processing container would have to have the complete ORM set up. I'd also need the two ORMs be completely alike. The other option is to use mongo inside the Rust container, but only have to create 1 ORM with Postgres setup. I chose the latter.

2 Technologies

2.1 Databases

2.1.1 PostgreSQL

PostgreSQL is this project's primary database. It's an RDBMS. It gets populated inside the data-population container. Due to the scope of this project, the PostgreSQL currently only contains three tables.

Like MongoDB and Meilisearch, this database contains a total of 112,923 entries

2.1.2 MongoDB

MongoDB is currently only used inside data-population. Information is extracted via. the Python pandas library and translated into a format suited for MongoDB. This information which is stored inside MongoDB is then extracted and used to populate both Meilisearch and PostgreSQL.

MongoDB was also intended to hold the changelog, but it has not been implemented in the current version of the program.

2.1.3 Redis

Redis is a NoSQL key-value database. It's used as a cache for the primary database, since calls to the postgreSQL database are expensive. It's an in-memory database, and is therefore not used for any tasks, where persistence is of great importance. At this point, the cache is cleared every 5 minutes for demonstration purposes. This can be changed in the container_values.env file.

2.1.4 Meilisearch

Meilisearch is a NoSQL search engine database. It has support for both Rust(backend) and Vue(frontend). There's also some quick setup functionalities for Digitalocean, which is a cloud infrastructure provider we've been using in the DAT education, which is the background most of the students have.

2.2 Backend

I've chosen to use Rust as my backend for this project. I've allowed myself to do so, because this is a single-man project. I also wanted to split the Data-Population container from the backend. Rust is a strongly typed, memory safe and very strict language. I feel more comfortable with using Rust as my backend language, than any of the others I know.

2.3 Data-Population

2.4 Frontend

3 Structure

4 Goals

This section is dedicated to the following assignment definition question:
Define functional and non-functional requirements to your project

4.1 Implemented

-

4.2 Planned

- Gathering of datasets
- Create diagrams for database
- data_population: Processing of datasets to usable format
- data_population: Datasets inserted into Mongo with Cython
- data_population: connection tests
- data_population: logging

- backend: Postgres, Redis, Mongo connectors and tests
- backend: Generate Postgres entries from Mongo database
- backend: Redis caching
- frontend: Basic site setup with routes
- frontend: Meilisearch functionality implemented
- frontend: Interaction with Redis routes.

4.3 Stretch Goals

- data_population: documentation with Sphinx
- backend: Documentation rustdocs

4.4 Scrapped

- Apply solution to Digital Ocean droplet
  - Create and setup server on digital ocean
  - docker-compose deployment version
  - CI with github actions.
  - github actions secrets with docker-compose
  - SSH SCP appleboy execution of docker-compose solution
  - nginx

5 Installation