NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Relatório P2 SCC

2024/2025

Group members:

Guilherme Antunes nº 70231
Guilherme Carvalhão nº 70735

Implementation Choices:

Application Specifications:

Our application is using 3 pods (users-shorts-logic, blobs-logic and Postgres-logic) and they substitute the azure services used in the 1º project. In this project we only have one specification related to the type of database (only PostgresSQL) and we didn't implement cache.

Our project organization:

Users-Shorts-logic:

This pod is used to received and answer the operations involving users or shorts or both.
In methods like the deleteUser, it needs to communicate with the blobs so that the blobs that have shorts from that user can be deleted.
Besides that, in simple methods like createUser, it needs to communicate with the Postgre's pod so that users can be created in the database.

Blobs-logic:

This pod is used to received and answer the operations involving blobs.
It receives operations like upload, deleteBlob, download and uses a persistent volume to store the data.

Postgres-logic:

This pod is used to received and answer operations that require interacting with the dataBase.
It receives operations like create, delete, update that are used to modify, add or remove elements from the dataBase.

Application Types of Data

Likes and Followers:

Similarly to our first project, we still have likes and followers. In this application, they are managed by the users-shorts-logic pods and are kept on the Postgres-logic one.
Besides that, we still have separate tables for likes and follows in the PostgreSQL.

Blobs:

In this project, the blobs are not from azure blob storage, instead we create a pod using a tomcat image. This pod is going to communicate with the users-shorts-logic one when needed or when called using the other container nodePort.

As we did on the first project, the server URI had to be hardcoded so that the blobUrl generated while creating a short would work.
We also changed thecopyWithLikes_And_Token method in the Short class to get the cookie, instead of the token, with the shortId.

PostgreSQL:

This time, our PostgreSQL is a pod based on a Postgre image from docker hub. However, we still create the same tables we did in our first project (users, shorts, likes, followers).

Azure IaaS:

In this project, we used the azure Kubernetes service to orchestrate our cointainers. Using azure Kubernetes facilitates the scalability of the application and logical division. However, as we will demonstrate in the application testing section below, the performance did not improve as anticipated; in fact, it slightly deteriorated.
Kubernetes is an excellent, powerful tool to manage complex applications that benefit a lot from a logical division, but in smaller ones, like ours, the augmented complexity it brings is not worth the possible benefits.

Application Tests:

We started by testing our application with postman, to verify if all the operations worked properly, and because it was easier to debug one request at a time. After we verified that everything was working as intended, we proceeded to test our application with artillery.

For the artillery tests we used the ones provided by Professor Kevin. We did the same tests we did for the first project and got the following results:

Min_time/ms	GetFeed	GetShorts	Followers	Likes
SQL/No Redis (1º proj)	73	84	69	67
SQL/No Redis (2º proj)	60	119	188	102

Max_time/ms	GetFeed	GetShorts	Followers	Likes
SQL/No Redis (1º proj)	109	98	69	104
SQL/No Redis (2º proj)	349	134	188	253

Mean_time/ms	GetFeed	GetShorts	Followers	Likes
SQL/No Redis (1º proj)	89	91	69	82.8
SQL/No Redis (2º proj)	154.9	124.6	188	177.5

As we can see our latency with Kubernetes is worse that a regular webapp due to Kubernetes added complexity.
In conclusion, Kubernetes could, however, be better if our application was being used at a much larger scale, as the ability to scale specific parts of the application horizontally would most likely be useful.