TuKano Implementation leveraging Kubernetes

Filipe Colla David and Victor Ditadi
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa

December 8, 2024

1 Introduction

This document explains the implementation of the TuKano App leveraging Kubernetes[3]. TuKano is a social network inspired in existing video sharing services, such as TikTok or Youtube Shorts. TuKano users can upload short videos to be viewed (and liked) by other users of the platform. The social network aspect of TuKano resides on having users follow other users, as the main way for the platform to populate the feed of shorts each user can visualize.

The implementation of the application is divided will be divided into several microservices that will be managed by a Kubernetes Cluster. The performance analysis of using the different services will be presented in Section 4.

2 TuKano Application

This application can be divided basically into three major logical components, Users, Shorts and Blobs.

The Users component holds all the logic for creating a user, deleting a user.

The Shorts component holds all the logic for creating, deleting a Short. A Short is a data model that holds the information for regarding a specific video, that was uploaded to the application. This component also contains the logic for following and liking other users/shorts.

The Blobs component holds all the logic for creating, deleting a blob.

These components are all related, however there's a stronger correlation between a Short and Blob, since every short contains the URL for a specific blob.

3 Architecture and Behavior

This application is contained inside a Kubernetes Cluster, which manages the deployment, scaling, and operation of the microservices. As mentioned in the introduction, the application was divided into several microservices.

3.1 Components, Microservices

As shown in Figure 3, there are 5 major logical components, they are all defined as microservices, except for the ConfigMap, which complements the functionality of these microservices, additionally each one has a service to allow them to be reached via a DNS entry, managed by the Kubernetes Control Plane, instead of using the private cluster IpAddress of that instance.

3.1.1 MINIO Service

Minio[4] is a high-performance, distributed object storage system. It is designed to handle large amounts of unstructured data, such as photos, videos, log files, backups, and container images, which makes it a perfect replacement for Azure Blob Storage. It has a PVC (persistent volume claim) attached to avoid losing any data in the event of a restart of this deployment and contains a LoadBalancer that exposes this microservice to the outside of the cluster.

3.1.2 PostgreSQL

This microservice serves as the main database of the application, where user, shorts, following, and likes information is stored. It is defined as a StatefulSet, which is common practice in the industry. It also has a PVC attached to prevent data loss in the event of a StatefulSet restart.

3.1.3 Redis Cache

This microservice is used to cache frequently accessed information and cookies for user authentication.

3.1.4 ConfigMap

This is an individual resource that holds all the necessary environmental variables for the application.

3.1.5 TuKano RestAPI

This is a microservice containing the central REST API of this application. It holds the logic for creating and removing users, managing shorts, and storing blobs.

This microservice, in its essential form, consists of a pod with a Docker container that runs the Docker image defined in the repository of this project. This Docker image is built on top of `tomcat:10.0-jdk17-openjdk-slim`, which allows it to run a Tomcat webserver that exposes a specific port for communication.

The communication with the database is handled using an Hibernate configuration and a JDBC Driver[6]. Communication with the MINIO storage is achieved through a MINIO Client[5]. Finally, uses Jedis[2], a Java client for Redis Cache, to interact with the caching service.

3.2 Automated Deployment

In order to easily deploy this application, a Makefile program was developed. A deployment of an application like this, requires several steps, and doing it manually is error prone. The process:

1. Compile the TuKano Application;
2. Build the docker image and push it to the DockerHub;
3. Update the docker image on the TuKano microservice yaml definition;
4. Deploy every microservice available.

As mentioned previously this process is mostly automated, however, there are some values from the ConfigMap that must be changed manually.

3.3 Behavior

Each microservice, upon starting will have access to the "Secrets" ConfigMap that holds all the necessary environmental variables for the configuration of their state. This includes the connections strings to the Database, Cache and Minio Storage, as well as some passwords and additional configuration parameters. However, as previously mentioned, there is a parameter on the ConfigMap that must be changed, the external URL of the MINIO LoadBalancer. Since the attribution of this URL is dynamic, this manual extra step is needed in order to ensure that the shorts will be stored with the right address to its blob. This is achieved by editing the "Secrets" ConfigMap, and deleting the TuKano RestAPI pod afterwards, so that it updates it's values from the new ConfigMap.

Once this initial configuration is complete, the application is ready to accept requests. As shown in Figure 3, the TuKano Rest API is the central communication of the whole ecosystem, it communicates with the Database and the Cache to store the relational information of the users activity, and communicates with the MINIO storage microservice to store the blobs. The MINIO makes this blobs available through a LoadBalancer that exposes the service to the outside of the Kubernetes cluster.

4 Performance Analysis

In this section there will be an analysis of the performance. The performance analysis will be done with the framework Artillery[1], that allows the programmer to test its application by creating several HTTP requests, once the experiment with the configured HTTP requests terminates, the framework outputs several statistics with the results obtained from the experiment.

The performance analysis will simulate a realistic flow of data over different configurations of the application:

- **Locally:** With Cache vs No cache, single instance of each microservice locally.

This scenario consists in making several HTTP requests to the application to test all endpoints, this are over the span of 10 seconds, starting with one client/second ramping up to five clients/second.

4.1 Locally

From the Figure 1 we can observe that both configurations, had the roughly the same amount of status codes, which is a good indication that both are well configured, additionally 2 we can see that using cache improves the response times, as excepted.

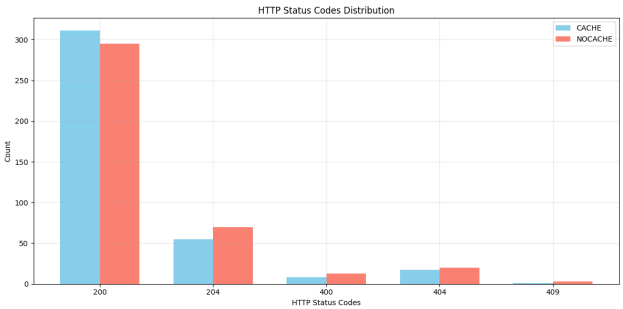

Figure 1: Performance Analysis Results

5 Future Work

As for future work, the main thing needed would be to change the ConfigMap that holds all the information for

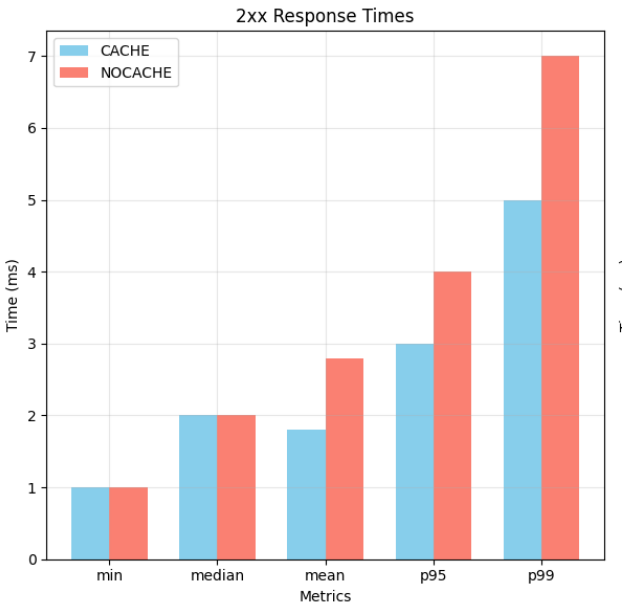

Figure 2: Response Times of

the microservices, including the passwords to a Kubernetes Secret, which much more secure for sharing sensitive information.

6 Challenges and Limitations

The project well when in deployed locally in Minikube, however some issues with persistent volumes and available nodes occurred when deploying to Azure IaaS.

7 Conclusions

When deploying microservices to Kubernetes, there is an initial definition of each individual microservice, when compared to SaaS, the code just needs to ready to connect to the already existing services.

However, development with Kubernetes is much faster using Azure SaaS, by using Minikube locally.

A Appendix

References

[1] Artillery. `https://www.artillery.io`, 2024. [Online; accessed 6/December/2024].

[2] Getting started with redis in java. `https://redis.io/learn/develop/java/getting-started`, 2024. [Online; accessed 6/December/2024].

[3] Kubernetes. `https://kubernetes.io`, 2024. [Online; accessed 6/December/2024].

[4] Minio. `https://min.io`, 2024. [Online; accessed 6/December/2024].

[5] Minio java api documentation. `https://min.io/docs/minio/linux/developers/java/API.html`, 2024. [Online; accessed 6/December/2024].

[6] Postgresql jdbc documentation. `https://jdbc.postgresql.org/documentation/use/`, 2024. [Online; accessed 6/December/2024].

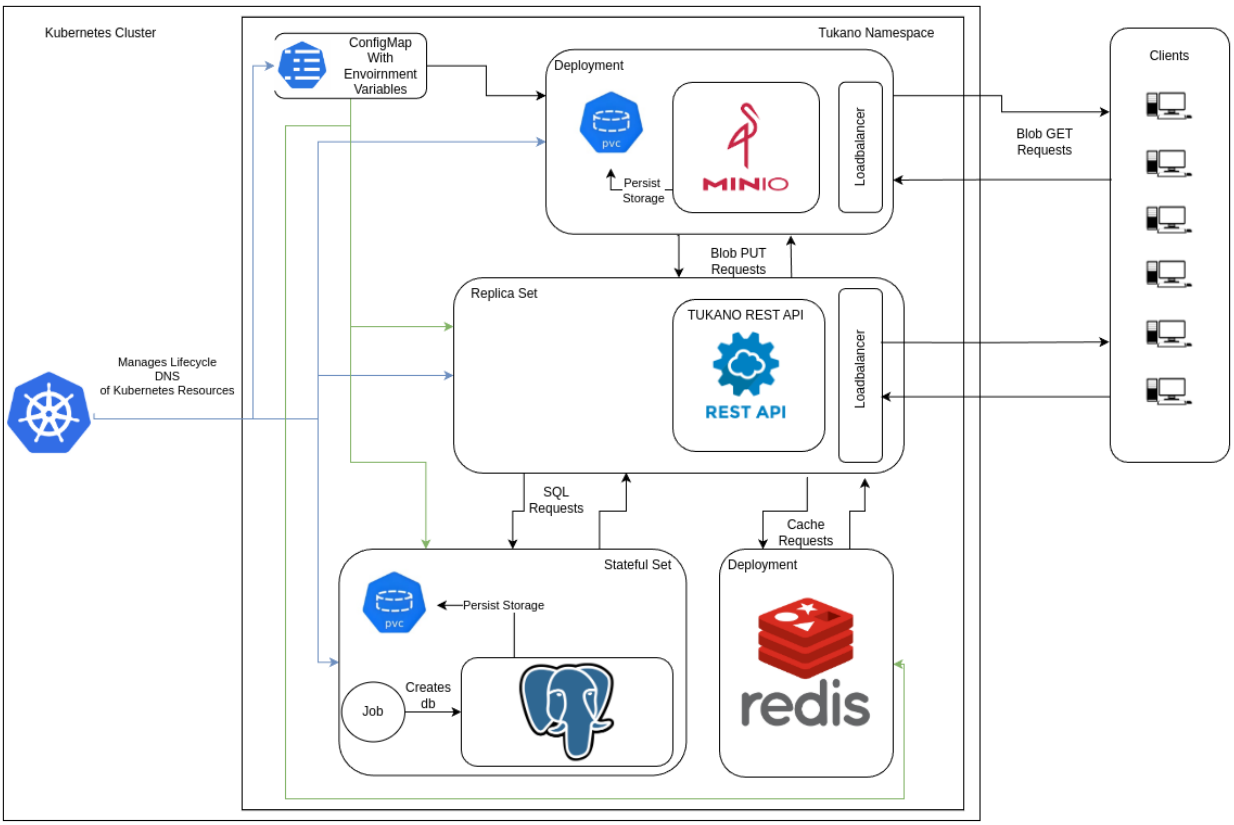

Figure 3: TuKano Application Architecture