Project Assigment 2 Report

Beatriz Machado 71863, Tiago Sousa 70624

26th of November 2024

1 Introduction

The present report details the transformation of the Tukano web application into a microservices-based architecture and its deployment to a Kubernetes cluster hosted on the Microsoft Azure platform. This shift to a containerized and orchestrated environment enables greater scalability, flexibility, and maintainability while taking advantage of Azure's infrastructure-as-a-service (IaaS) capabilities.

2 Migrating the Application to Microservices

When deciding how to divide our application into microservices, we considered the deployment environment in Kubernetes and aimed to optimize its automatic scaling capabilities. To achieve this, we focused on dividing the application based on the network throughput and request requirements of each component. This approach led us to create four distinct microservices: a Redis cache microservice, a PostgreSQL database microservice, a blob storage microservice, and the application microservice, which serves as the central gateway for all requests. This division allows each service to scale independently based on demand, ensuring greater efficiency and flexibility.

2.1 Redis Cache

To migrate the application cache to the new microservices-based architecture, we utilized the Redis container provided by Docker. The container was launched with a password passed as a command-line argument in order to prevent unauthorized access. Within the Tukano application, the cache configuration was updated to reflect the Redis container's hostname and the password set during container creation, ensuring a seamless integration and secure communication between the application and the deployed Redis cache.

2.2 PostgreSQL DB

To migrate our application's database, we preserved the existing SQL implementation by utilizing the PostgreSQL container provided by Docker. The container

was launched with a specified username, password, and database name, passed as environment variables during setup. The Hibernate configuration XML was then updated to include the container's hostname and the previously specified credentials. This approach ensured a smooth transition while maintaining the application's existing database structure.

2.3 Blob Storage

In order to minimize the code changes necessary to migrate Tukano's blob storage, we implemented it as a separate microservice. This microservice exposes three endpoints: one for uploading blobs, one for downloading blobs and another for deleting blobs. When the Tukano application receives a blob-related request (upload, download or delete), it performs the necessary validations and proxies the request to the blob storage microservice, returning the response to the client. This design ensured minimal alterations to the existing codebase, requiring only the implementation of a REST client in the Tukano application to interact with the microservice. To protect the blob storage microservice from unauthorized access, we employed a shared secret authentication mechanism. Only requests containing this secret, which is exclusively shared with the Tukano application, are authorized, providing a secure and efficient solution. After implementing the blob storage microservice, a simple Docker container was created that launches and runs the microservice.

2.4 Tukano Application

For the Tukano application itself, we created a Docker container that launches and runs the application. This container encapsulates the entire application and acts as the central gateway for all requests, interfacing with other containers as needed.

3 Local Testing

To test the functionality of the application and its interaction with each microservice, we adopted an incremental approach. This involved testing each microservice integration in isolation before gradually adding others. To facilitate this, we provided alternative implementations for each microservice that work locally. These local implementations allowed us to test communication and integration with each service in isolation. The specific implementation used is determined by an environment variable. For instance, we could launch the Tukano application with a local cache and local database, while having it communicate solely with the blob storage container.

For local testing, we created a Docker Compose file to deploy each microservice as a container. This setup enabled us to easily debug each microservice and their interactions locally, providing a more controlled environment compared to testing in a Kubernetes cluster. This approach ensured that we could isolate

issues and validate each microservice's functionality before deploying them in the final environment.

4 Kubernetes Cluster Migration

For the migration to Kubernetes, we created deployments that mirrored the configurations already present in our Docker Compose setup. We wrote YAML files for each microservice, which included two key components: a deployment file and a service file. The deployment file describes how to launch the container within each pod, including any necessary environment variables, while the service file handles networking and communication between the pods.

Each pod was configured to internally expose the ports used by the containers running inside it. The exception to this was the Tukano application pod, which needed to expose an external port to allow clients to communicate with the gateway. For environment variables that required sensitive information, we utilized Kubernetes secrets to ensure secure management.

4.1 Blob Storage Pod

For the blob storage microservice, we added an extra layer of functionality by creating a persistent volume. This volume was mounted at the path where blobs are stored, ensuring that the blob storage remains persistent even if the pod is deleted or recreated. This approach allowed us to retain blob data across pod restarts, providing a reliable storage solution in our Kubernetes environment.

4.2 Deployment Process

The deployment process began by testing the application in a local Kubernetes cluster using Minikube. The deployment involved the application of all previously created YAML files. After verifying that the local deployment was functioning correctly, we proceeded with the deployment to Azure. This involved changing our *kubectl* configurations to point to our Azure account and applying the YAML files. This incremental approach ensured smooth deployment and easy troubleshooting at each stage of the process.

5 Performance Evaluation

5.1 Observations

During the performance evaluation of our Tukano application, both the Azure Kubernetes cluster deployment and the Azure PaaS deployment exhibited similar request latencies under identical conditions. This uniformity persisted across various scenarios, due to Azure's optimized internal network infrastructure, which minimizes latency differences for applications deployed within the same region.

5.2 Geographical Impact

A significant difference in latency was observed when deploying the application in a geographically distant region. For example, when the application was deployed in the West US region, the request latency increased substantially. This is attributed to the inherent delay in long-distance network communication, highlighting the importance of deploying applications closer to end users for latency-sensitive workloads.

5.3 Hypothetical Scenarios

While the current environment did not reveal significant performance differences, the Kubernetes deployment offers a robust foundation for handling stress conditions and scaling scenarios. For instance, the independent scalability of microservices allows us to scale specific components, such as the blob storage service, to handle a high volume of requests while keeping the costs of other microservices low. This level of fine-grained, automatic scaling was not achievable in the previous PaaS deployment. Additionally, Kubernetes provides self-healing capabilities, enabling the system to automatically detect and recover from failures, ensuring greater resilience and reliability in production environments.

The Kubernetes deployment also excels in efficiency and ease of deployment. By using Kubernetes, the application requires no additional configuration during deployment, ensuring a consistent and reliable environment. This streamlines the deployment process while enhancing the portability of the application across different infrastructures.