Project Report: Adaptation and Deployment of Tukano on Azure Kubernetes Service (AKS)

Rodrigo Silva Nº70567
António Salselas Nº70568

8th December 2024

Introduction

The objective of this project was to adapt and deploy the Tukano application, originally developed during the first assignment, into a Kubernetes-based infrastructure hosted on Azure. This initiative required transitioning from the Azure Platform-as-a-Service (PaaS) solutions used previously to an implementation leveraging Azure Kubernetes Service (AKS). The primary challenge was to ensure that all of the application's original functionality remained intact while optimizing it for containerized environments.

To achieve this, we utilized Docker as the core tool for creating and managing containerized images of the application. These images provided a consistent and portable runtime environment, ensuring the application could run reliably across various stages of the deployment lifecycle. Kubernetes was employed to handle the orchestration and scalability of these containers, enabling efficient resource utilization and simplified management of multiple application instances.

The solution involved leveraging existing container images available on Docker Hub, as well as creating customized images tailored to the specific requirements of the Tukano application. This customization ensured that the application was fully compatible with the new infrastructure and adhered to performance and security best practices. By integrating Docker and Kubernetes, we were able to create a robust, scalable, and highly available system while laying the groundwork for future enhancements and seamless scaling in a cloud-native environment.

Detailed Steps to Implement the Project

1. Tukano Application

The first step was to containerize the Tukano application to make it compatible with a Kubernetes-based infrastructure. Before deploying to the Azure Kubernetes Service (AKS), we tested the application locally using Minikube, a tool for running Kubernetes clusters on a local machine. Minikube allowed us to verify the deployment and service configurations. After testing, we confirmed that the application worked as intended.

- Dockerfile Creation: We wrote a Dockerfile to package the Tukano application into a Docker container. The Dockerfile included all dependencies, configurations, and

environment variables required to ensure the application runs consistently in any containerized environment.

- Kubernetes YAML Configuration: To deploy the Tukano application on Kubernetes, we created two YAML files:
  - A Deployment YAML file, defining the pod specifications (such as container image, replicas, and environment variables) and ensuring the application could scale horizontally.
  - A Service YAML file, configuring how the application could be accessed by other resources within the cluster or by external clients.

2. PostgreSQL Database

Next, we migrated the application database from CosmosDB to PostgreSQL, running within a containerized environment on Kubernetes.

- Database Selection: We chose PostgreSQL as the relational database due to its reliability and compatibility with the Tukano application. An official PostgreSQL Docker image from Docker Hub was selected to ensure stability and best practices.
- Kubernetes YAML Configuration: Similar to the application deployment, we created two YAML files for PostgreSQL:
  - Deployment YAML file, specifying the PostgreSQL container image, resource allocation, and necessary environment variables (e.g., database username, password, and name).
  - A Service YAML file, enabling the application server to connect to the PostgreSQL instance within the Kubernetes cluster.

3. Persistent Media Data Storage

To ensure the durability of media files uploaded by users, we replaced Azure Blob Storage with a Kubernetes-based persistent storage solution.

- Filesystem Modification: The Tukano application was updated to write media files using a persistent storage backend instead of a cloud-based solution. This change ensured that uploaded files would not be lost even if the container was restarted or replaced.
- Persistent Volume Configuration:
  - We configured a Persistent Volume (PV) to allocate storage space for the media files. Azure Disk was used as the backend to provide reliable and scalable storage.
  - A Persistent Volume Claim (PVC) was created to request storage from the PV and bind it to the application pods. This binding ensured that all instances of the Tukano application had access to the same durable storage location.
- Kubernetes YAML Update: The Deployment YAML file for the Tukano application was updated to include the PVC as a volume mount, enabling the application to store and retrieve files from the persistent volume seamlessly.

4. Redis Cache

To optimize the application and provide caching capabilities, we added Redis to the Kubernetes environment.

- Redis Image Selection: We selected an official Redis Docker image from Docker Hub for deployment. Redis was chosen due to its lightweight, high-performance capabilities and compatibility with the Tukano application's caching needs.
- Kubernetes YAML Configuration: Two YAML files were created for Redis:
  - A Deployment YAML file, configuring the Redis container image, resource allocation, and deployment settings.
  - A Service YAML file, enabling the Tukano application to connect to Redis within the Kubernetes cluster.

5. Cookie-Based User Session Authentication

The final step was implementing user session authentication to secure access to media files stored in the persistent volume.

- Authentication Logic Implementation: We implemented cookie-based user session authentication for the Tukano application's media service. The authentication mechanism was built on the code from Lab 9, adapting it to meet the project's specific requirements.
  - The JavaBlobs Class was altered in order to handle user login, validate credentials, and manage session tokens securely.

Performance Tests

During the initial stages of the project, we utilized Azure Kubernetes Service (AKS) for testing the implementation of our solutions. However, due to the credits consumed during the first project and the associated costs of running AKS for extended periods, we were unable to conduct comprehensive performance tests on the AKS platform. This limitation necessitated an alternative approach to validate the performance and scalability of our application.
To overcome this constraint, we turned to Minikube, a local Kubernetes environment, for running our performance tests. Minikube provided a cost-effective and efficient solution for simulating a Kubernetes cluster on our local systems. Although Minikube cannot perfectly replicate the scalability and production-level performance of AKS, it served as a practical environment to benchmark different application configurations and evaluate their relative efficiency. We designed our performance tests to evaluate the impact of caching on the Tukano application's overall performance. Two configurations were tested:

- **Without Redis Cache:** In this setup, the application operated without utilizing any caching mechanism. Every data request was processed directly through the database, simulating the baseline performance of the application.

- **With Redis Cache:** In this configuration, we integrated Redis as a caching layer. Frequently accessed data was stored in Redis, allowing the application to retrieve it from memory instead of querying the database. This setup was intended to demonstrate the potential performance improvements achieved by introducing a caching mechanism.

In the following tables, we present our results. Its important to highlight that the sample size is small, due to the same error presented in the first project. All the times are in milliseconds.

Endpoint	Min	Max	Mean	Median	P95	P99
Create User (Post)	13	37	19.6	16.9	29.1	29.1
Create User (Put)	11	63	19.9	13.1	32.8	32.8
Get User	4	22	7.1	6	6	6
Upload Short	6	54	14.3	7.9	27.9	27.9
Create Short	8	62	19.1	13.9	18	18

Figure 1: Application performance with Redis Cache.

Endpoint	Min	Max	Mean	Median	P95	P99
Create User (Post)	8	16	12.2	10.1	15	15
Create User (Put)	7	19	10.5	10.1	10.9	10.9
Get User	4	6	5.1	5	6	6
Upload Short	5	9	6.8	7	7	7
Create Short	11	293	41.6	13.1	18	18

Figure 2: Application performance without Redis Cache.

Conclusion

The Tukano application was successfully migrated to Azure Kubernetes Service (AKS), replacing the previous Azure PaaS solutions with Kubernetes-based alternatives. This transition involved the seamless implementation of the application server, database, cache, and persistent storage, all of which were designed to ensure the scalability, security, and reliability of the system. However, there were a few challenges during the project that need to be highlighted. First, due to an oversight on our part, we ran out of Azure credits, which prevented us from conducting deployment tests directly on the Azure platform. As a result, testing was carried out locally using Minikube to simulate the AKS environment. Additionally, the GitHub commit history may appear unevenly distributed. This is because we used the "Code With Me" feature in IntelliJ IDEA for collaborative work, which resulted in fewer individual commits being recorded. Despite these constraints, the migration project successfully demonstrated the feasibility of deploying the Tukano application on a Kubernetes-based infrastructure. It also highlighted the significant advantages of containerization and orchestration, showcasing how these modern technologies can enhance the scalability, flexibility, and overall efficiency of application development.