Cloud Computing Systems

2024/2025

Tukano Project

Project Assignment #2

Authors:

70070, Pedro Peralta
71794, Tsimafei Nestsiarau

Professor:
Sérgio Duarte

December 6, 2024

Introduction

This report describes the process of deploying an application within a Kubernetes environment. The main goal of this project was to transition from a cloud-based infrastructure that relied on Azure resources to a locally managed Kubernetes setup. To achieve this, we used Minikube for local deployment, created custom Docker images for each service, and used Kubernetes to manage and organize the different components of the application.

Our application consists of three main services: Users-service, Shorts-service, and Blobs-service. Each service has a specific role, such as managing users, handling short, or storing “blobs”. These services were deployed in separate pods, with each pod running a single container. This approach ensures that the services are independent, making the system more reliable and easier to scale if needed.

To manage data, we replaced Azure’s database services with a PostgreSQL database hosted inside the Kubernetes cluster. The database was carefully configured to integrate with the application and support all operations.

For file storage, we replaced Azure’s Blob Storage with Kubernetes Persistent Volumes (PV). This allowed us to set up a reliable and flexible storage system that the Blobs-service could use.

User authentication and session management were also implemented using cookies. These features allow the application to handle user sessions securely and efficiently.

The deployment was managed using a set of YAML configuration files, which defined all the necessary resources such as storage, services, and deployments.

Although the project has made significant progress, there have been challenges along the way, especially in configuring the Kubernetes components. Due to these difficulties, it was not possible to carry out load and performance tests with Artillery. Lack of study material made the process of configuring and managing Kubernetes services complex.

There are two branches in the repository that contain two solutions, which is one of the ways we've come up with to show our progress. One of them is “**Kubernetes**”, which contains all the configurations needed to run user, shorts and blobs services. Users and shorts are stored in-memory with the usage of HashMaps; we've used ChapGPT to configure in-memory storage in JavaShorts and JavaUsers classes . We were able to interact with users but there’s some unknown issue with shorts. The persistent volume is working, we managed to save some files inside a container and these files were stored in both container’s storage and persistent volume. Authentication works as well. In general, the only part that is missing is the database. We couldn’t configure the PostgreSQL database to work with the implemented functionality. The other branch is “**Kubernetes2**” which contains the implementation and changes to make postgres boot successfully.

Deploying to Kubernetes

In this assignment, we have deployed our application locally with Minikube. In order to perform that, we have installed Minikube, kubectl and configured 3 docker image for each service :

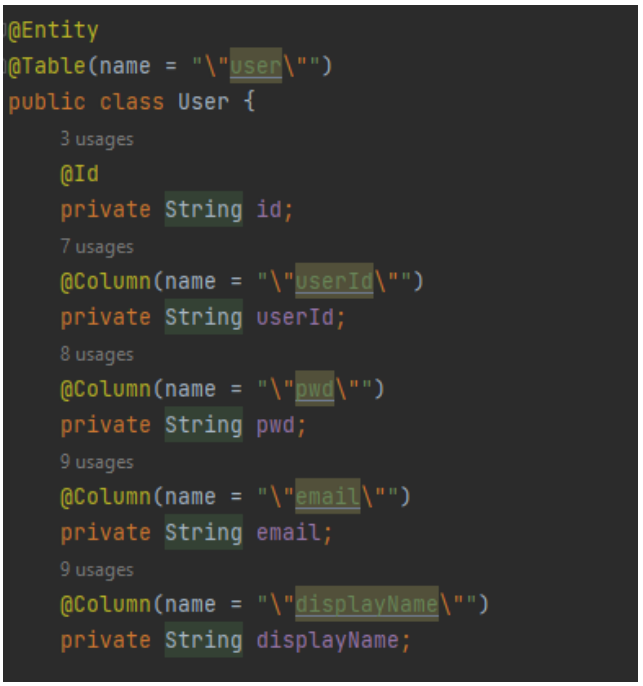
- Users-service for handling operations with users
- Shorts-service for handling operations with shorts
- Blobs-service for replacing Azure's Blob Storage and handling operations with files

Each service is running under its own pod with a single container (wrapper pod). Splitting services provides better reliability in case one service fails and future scalability. Every pod is deployed with the usage of custom Docker images and every container has only a single replica for simplicity and better performance.

Database

In our project, we used a PostgreSQL database to store and manage data to replace the resources we had previously used from Azure. To ensure integration with our application, we created and configured a PostgreSQL service in Kubernetes. We started by implementing the PostgreSQL configurations, indicating which ports we would use and the DB access data. Once we had the DB configured, we moved on to configuring the dedicated postgres service, configuring it for the same port as described above.

We had to make a change to the User class because the word user is a reserved word in the context of postgres, so we ended up inserting extra quotation marks so that it wouldn't be a reserved word. You can see in the following image how it was implemented.



User Class implementation

Persistent Volume

Azure's Blob Storage was replaced with files storage that uses simple Java.io.File API. All files (blobs) inside the blobs-service container are mounted to a path that is mapped to the Persistent Volume and vice-versa. Blobs-service interacts with persistent volume via pvc and has ReadWriteOnce access mode. We have allocated 1Gb of memory for a PV which should be enough (if not too much) for test purposes.

YAML Structure

In the following table we have our yaml files to configure the application:

branch: Kubernetes

Redis Cache	redis-service.yaml
Services	tukano-aks.yaml
Persistent Volume	tukano-pv.yaml
Persistent Volume Claim	tukano-pvc.yaml
Postgres Database	postgres-deployment.yaml

branch: Kubernetes2

Storage	tukano-storage.yaml
Persistent Volume Chain	tukano-pvc.yaml
Persistent Volume	tukano-pv.yaml
Postgres Database	postgres-deployment.yaml
Application Deployment	deployment.yaml
Application Service	tukano-service.yaml

To start all the services in our application, we run the command **kubectl apply -f *<yaml_fileName>*** in the same order as in the table above

User Session Authentication

To implement authentication in our application, we used the code provided in the lab class as a basis, making the necessary adaptations to integrate it into our application.
This system is a basic implementation of authentication and session management for our REST application using the Jakarta API. We use cookies to manage users' sessions.

We have the following classes with their respective responsibilities:

RequestCokkies	Stores the cookies of each request using ***ThreadLocal***, ensuring isolation between threads
RequestCookiesFilter	This is a request filter (***ContainerRequestFilter***) that runs before the request is processed
RequestCookiesCleanupFilter	Clears cookies stored in ***ThreadLocal*** to free up memory and avoid data leakage problems between requests.
Session	Represents an authenticated user session.
FakeRedisLayer	Simulates a database (**Redis**) to store user sessions.
Authentication	It implements the user authentication flow, managing login and session validation.

To log in, the user accesses the "**Login.html**" html and fills in the characters. In our development, as the priority was not to implement a login system but only to present the implementation of cookies, any username and password will be valid. There is a username which, if used, will be identified as admins and will have such permissions.

Here we have the username of the admin user:

```
2 usages
private static final String ADMIN = "pedro";
```

Authentication Class

Within this class we have two methods that are used to validate the user's session (***validateSession*** and ***isLoggedIn***).

The ***validateSession*** method has not been changed, as it came from the lab code. This method checks if there is a cookie present, and if there is, it checks if the name of the user inside the cookie is the same as the one being passed in as a parameter. This method is mainly used when we want to validate that the user trying to perform a task is admin.
The ***isLoggedIn*** method was created by us so that no username is required for the user to be considered logged in. This method only checks if there is an authentication cookie, if there is then the user is considered logged in.

```
3 usages   PedroFPeralta
static public Session isLoggedIn() throws NotAuthorizedException {
    var cookies = RequestCookies.get();

    Cookie cookie = cookies.get(COOKIE_KEY);

    if (cookie == null )
        throw new NotAuthorizedException("No session initialized");

    var session = FakeRedisLayer.getInstance().getSession(cookie.getValue());

    if( session == null )
        throw new NotAuthorizedException("No valid session initialized");

    if (session.user() == null || session.user().length() == 0)
        throw new NotAuthorizedException("No valid session initialized");

    return session;
}
```

isLoggedIn Method Snippet code

Artillery Tests

Artillery is a powerful tool for carrying out load and performance tests on APIs and web applications. In the context of this project, using Artillery tests could help ensure that the application and associated services, such as the PostgreSQL database, are able to handle different volumes of traffic efficiently.

These tests simulate multiple users accessing the application simultaneously, making it possible to identify performance bottlenecks, validate scalability and guarantee a consistent experience for end users. It is essential to deliver a reliable system, even in conditions of high traffic or unexpected loads.
In this second part of the project, we ended up not implementing these tests due to the many problems we encountered throughout the kubernetes implementation process.