Cloud Computing Systems - Azure Tukano in Kubernetes Cluster

Gonçalo Mateus[60333] and Rodrigo Grave[60532]

NOVA School of Science and Technology, Almada, Portugal
https://www.fct.unl.pt/en

1 Project implementation

1.1 Implementation Process

In this project we based ourselves on the lab classes and we reused some of the code from the previous project in the way to implement Redis and PostgreSQL.

1.2 Challenges and Decisions

We ran into various different challenges during the implementation

- **Deploy on Azure Kubernetes Service**: We based ourselves in the labs to realize this task, we tried to adapt the yaml file to our context and we deployed to Azure successfully. The adaptation process of the yaml file was a bit tricky and complicated, we needed to study and search to really know how to create all the services we needed in the correct way.

- **Creating PostgreSQL DB service in the cluster**: To create this service in the Kubernetes cluster, we added in the cluster configuration a new service that runs a PostgreSQL image from Dockerhub. We had to search for the environment variable names that are used inside the image to be able to configure Hibernate to correctly connect to the service (the variables were `POSTGRES_USER` and `POSTGRES_PASSWORD`).
  For the code we based ourselves on the base project given by the teachers and had to make some adjustments to the queries to be able to run them in the PostgreSQL service.

- **Blob Storage using a persistent volume**: In order to have a storage for the blobs, a persistent volume inside the cluster was created. We needed to create both a `PersistentVolume` (that is the creation of the actual volume) and a `PersistenVolumeClaim` (that is used by the Tukano service to claim what it needs from that persistent volume). Then a mount is added to the Tukano service that points to the persistent volume. This mount is seen in the Pod where Tukano is running as a local folder that it can write in the same way as writing in the local system. For writing the blobs we also bases ourselves in the base project, wich used the `FileSystemStorage` class to manage the file writing and reading in the file system.

- **User Session Authentication**: To realize this task we decide to implement a login endpoint instead of integrating it on create user and get user endpoints. We decided that to avoid changing the endpoints code and to make it more organized. We used the filters as proposed in the Lab class, and we decided to store the cookies in Redis with a expiration time equal to the cookie's one so that way we could easily know if the cookie expired. We added an Admin verification too so only the user with the username "Admin" can delete blobs. We decided that only the owner of the short can realize the upload of the respective blob, and everyone can download blobs if they have a valid cookie. The error `UNAUTHORIZED` is returned when a cookie is not valid.
  - **NOTE**: the login endpoint is: <URL>/login/{userId}?pwd=<passowrd> password is a queryParam and userId is a pathParam.

- **Redis Cache**: In the implementation of Redis we based ourselves in the previous project solution. We used a write through strategy making cache of Users, Shorts, Shorts like list, Users followers list, Users shorts list, a counter for shorts likes and the cookies. The implementation of Redis is very similar to the previous one with the addiction of cookies caching. We added too a environment variable `HAS_CACHE` that when set to "true" the system uses all caching features and being something different it will only use cache for the cookie, being essential for the working of this application.

1.3 Performance Testing and Analysis

In this section we tested our application with the help of Artillery scripts for load testing. Some of them created by us and the others given by the teachers. Artillery runs various requests in sequence and gives tools to simulate basically any type of usage behavior. In the end of the tests, Artillery presents a report on the average overall response times and the response times of each different type of request/operation. In the following tables we used the values min max and mean of the summary report of each artillery test. We ran the tests several times and we decided to show the values that we thought would be more adequate. In this project testing we focused on the impact of the usage of Redis in the application, so the following tests will be displayed as Tables with a Cache and No Cache column. The first row is the name of the artillery test file we used to obtain the results on its row.

Our Tests:

	No Cache	**Using Cache**
User_Register	min: 59 max: 3269 mean: 108.1	min: 58 max: 180 mean: 78.4
Upload_Shorts	min: 71 max: 155 mean: 90.2	min: 69 max: 183 mean: 86.8
Realistic_Flow	min: 67 max: 175 mean: 86.2	min: 63 max: 156 mean: 82.6
Shorts_Test	min: 66 max: 105 mean: 82	min: 70 max: 113 mean: 82.1

Table 1. Artillery test results(in milliseconds)

– **Analysis of using Redis Cache**

Analyzing our results on the tests we performed we can conclude that we achieved the expected results. The table show us in milliseconds the longest request, the shortest and the mean of all the requests performed in the relative test. Being all the mean values or very similar or really better in "Using Cache" we can really say we achieved the results we were expecting. Due to the fact that cache is responsible for storing the most recent used data and to guarantee fastest data access compared to the use of a Database. Comparing max values, observing User_Register and Shorts_Test we can see that the cache strategy gave us higher results. Se interpreted that as being fault of the write through technique we used for caching. That technique is really good for data reads, but slightly worse for data writes, because in the writing methods the server stores the data on Redis Database increasing a little bit the throughput. This way, the worse max values in the Using_Cache strategy are justified.
In sum we got the expected results, using a write through strategy. But we think that with more time to analyze and realize the project we could have implemented a better caching strategy and extend the tests to get more conclusions.

1.4 Comparison With the Previous Project

The results obtained were far more positive than the ones obtained in the previous project when comparing the usage of cache. In the previous project (deployment in Azure using different Azure services) we obtained generally worse results when using a cache compared to when not using one.

This time, we either got very similar or better results when using the cache, which, as mentioned above, is what we expected. We give this difference to the fact that the different Azure services (particularly the Redis Cache) were in different regions, which might have caused higher latency. In the Kubernete's case, all services, pods and persistent volumes are running in the same container cluster, meaning that the latency will not have near as much effect as it has in the first project.

The Geo-replicated solution, as expected, have really worse values compared to this one. We give that to the fact that with Geo-replication, the services need to communicate with more than one region, generating more response time and throughput.

2 Conclusion

In conclusion, we consider we have done a good job implementing the Azure Kubernetes Service, having had quite good results on the tests performed. We enjoyed a little bit more this project, and we found the using of Kubernetes more interesting comparing too the previous project. We could not achieve all the optional requirements due to the fact that we did not implement Azure functions in the previous project, and we didn't have much more time to deploy the system in Azure Containers. We think the tests we realized are enough to get some conclusions but we could extend them to more requests in the way to try to see some different result and get better conclusions.