Cloud Computing Systems - Azure Tukano

Gonçalo Mateus[60333] and Rodrigo Grave[60532]

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

IMPORTANT NOTE: During the tests we noted that we had an error on the PostgreSQL code, a simple line that makes it impossible to run the PostgreSQL on Azure. The way that we have it only allows to run PostgreSQL on tomcat local server. So we suggest that, to make it possible to run in Azure, remove in "`src/main/java/utils/Hibernate.java`" the "`new File(HIBERNATE_CFG_FILE)`" inside the .configure() in Hibernate() constructor.

1 Project implementation

1.1 Implementation Process

In this project we based ourselves on the lab classes to adapt the given base code to leverage Azure Web Services so that the code was able to run in the cloud platform instead of running locally.

1.2 Challenges and Decisions

We ran into various different challenges during the implementation

– **Azure Blob Storage**: the intent was to store the blobs (byte stream containing the actual video) in a storage account in azure. Originally the blobs where being stored in the file system of the local machine and they where organized by directories, each corresponding to a different user. The challenge was to understand how Azure Blob Storage deals with directories. We came to the conclusion that Azure Blob Storage does not have the notion of directories even though it shows the blobs like a file system when the blobs names contain a '/' character.
This meant that for the deleting part of the blobs when deleting a user we cannot simply delete the entire directory with one operation, but instead have to loop through all the blobs, deleting only the ones that contain the deleted user's name in their id.
To make it easy to understand and debug we decided to, before putting the `blobId` in the Azure Blob Storage, change the "+" character that splits the `userId` from the rest of the `blobId` to a "/", so that way Azure Blob Storage shows a directory of user's blobs.

- **Azure CosmosDB**: to store the users and shorts information in a more organized manner hibernate was being used to store this information in a local relational data base using Hibernate. We then implemented 2 types of databases available in the Azure Cosmos service so that we could compare their performances:

  1. **CosmosDB NoSQL**: for NoSQL we ran into a challenge related to the id of each entity. We opted to create a container for each type of entity (**users**, **shorts**, **following** and **likes**) but NoSQL requires every entity to have one property with the name "id". We overcame this issue by adding a tag (`@JsonProperty("id")`) above each entity's id property so that the actual request body contained this required property.
     Another challenge was that the code given to us in the labs (`CosmosDBLayer`) only stores the information in one container, being initialized in `CosmosDBLayer`. To overcome this issue we created a client for each of the containers in the respective server class (Users container in `JavaUsers`, and the rest in `JavaShorts`). This way the respective container passes to the `CosmosDBLayer` class as an argument of all methods that do the persistence in the NoSQL DB.
     Finally, when implementing the functions logic, we get into other issue. The previous tukano implementation was on hibernate, that used an SQL language rather than NoSQL that contains a very limited query language, so we needed to change some logic of functions like `getFeed`, and the majority of the deletes. In the deletes, given the fact that NoSQL, does not support DELETE clauses, we needed to first perform a get with the SELECT clause and then finally delete the information. In `getFeed`, given the fact that NoSQL does not support JOIN clauses and transactions between containers, we needed to split all the SELECTS and perform some for cycles to make sure that the function returns what we want.
     After this issue, we concluded that probably a better way of doing the persistence in NoSQL was to not store the entities in different containers.

  2. **CosmosDB for PostgreSQL**: for PostgreSQL we took advantage of already having the hibernateDB implementation and, with only the change of the `hibernate.cfg.xml`, it was possible to make it use the CosmosDB Azure service.

- **Azure Redis Cache**: the use of cache is essential for this project in order to have a better performance on operations that are executed a large amount of times. The challenge in implementing the cache was to choose the strategy for caching. We opted for a write-through cache both in writes and reads, also meaning our cache will always be with correct information. On our writes, after store the information on the DB, we put it on cache. On our reads, if the information is in the cache, we just return it, if not we get it from the DB and then store it on the cache.

For `createUser/getUser` and `createShort/getShort` we set them normally into the cache; for `deleteUser/deleteShort` we delete them from the cache; for `followers` the list of followers of a user is stored in cache; for `follow` the follower's id is either stored on or removed from the list of followers of the followee; for `likes` a list of users that liked the short is stored in cache; for `like` the user is stored on or removed from the list of likes from the corresponding short as well as incrementing or adding a counter that stores the current number of likes of that short.
Given the fact that Redis is a key-value store, we decided to define the keys with a format `"prefix:id"` so that prefix is a static string that identifies the kind of value we are storing, and id is the id of the entity being stored.
After the implementation of Redis, we got to the conclusion that we defined a really complex strategy for caching, and that cost us many of the time we had to do the project.

1.3 Advanced features implementation

- **Geo-Replication support**: currently our project is able to replicate blobs storage and database (CosmosDB). The replication of the database is automatically setup in `AzureManagement` for the regions that are chosen, meaning that for the database only one resource is created in Azure.
  For the blob storage, `AzureManagement` creates the same amount of storage accounts than regions chosen. In our case we are running with 2 regions (North Europe and Japan East). Because we are not yet using Kubernetes for secrets and keys management, we adapted `AzureManagement` to create the blob storage account keys in the props files with different names so that the keys would not override when loading the props sequentially.

  NOTE: AzureManagement creates the necessary resources and stores the access keys in `src/main/resources` package.

1.4 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.

With Geo-Replication

Our tests:

	NoSQL	**PostgreSQL**
User_Test	min: 120 max: 3341 mean: 348.4	min: 100 max: 4005 mean: 382.1
Shorts_Test	min: 120 max: 1404 mean: 341.4	min: 86 max: 369 mean: 162.4

Table 1. Artillery test results **with cache**

	NoSQL	**PostgreSQL**
User_Test	min: 83 max: 262 mean: 98.9	min: 66 max: 528 mean: 106.1
Shorts_Test	min: 100 max: 1955 mean: 240.8	min: 73 max: 2729 mean: 273.7

Table 2. Artillery test results **without cache**

Teachers tests:

	NoSQL	**PostgreSQL**
User_Register	min: 112 max: 572 mean: 182.9	min: 69 max: 2057 mean: 246.7
Upload_Shorts	min: 184 max: 3998 mean: 1228.5	min: 236 max: 2979 mean: 1295
Realistic_Flow	min: 150 max: 584 mean: 300.4	min: 99 max: 1712 mean: 399
User_Delete	min: 155 max: 8224 mean: 809.3	min: 123 max: 2931 mean: 377.4

Table 3. Artillery test results **with cache**

	NoSQL	**PostgreSQL**
User_Register	min: 80 max: 9185 mean: 692.9	min: 63 max: 9613 mean: 660.2
Upload_Shorts	min: 152 max: 1269 mean: 606.6	min: 182 max: 3421 mean: 754.8
Realistic_Flow	min: 99 max: 957 mean: 181.5	min: 79 max: 764 mean: 125.6
User_Delete	min: 107 max: 9689 mean: 1506.6	min: 76 max: 9309 mean: 821.7

Table 4. Artillery test results **without cache**

Writing/reading in only one region

	NoSQL	PostgreSQL
User_Register (not replicated)	min: 107 max: 3806 mean: 186.6	min: 103 max: 5294 mean: 206
User_Register (2 regions replicated)	min: 112 max: 572 mean: 182.9	min: 69 max: 2057 mean: 246.7

Table 5. Artillery test results **with cache**

	NoSQL	PostgreSQL
User_Register (not replicated)	min: 77 max: 3828 mean: 199.7	min: 62 max: 4331 mean: 220.1
User_Register (2 regions replicated)	min: 80 max: 9185 mean: 692.9	min: 63 max: 9613 mean: 660.2

Table 6. Artillery test results **without cache**

– **Analysis of using NoSQL vs PostgreSQL**

Analyzing our results on the tables on the purpose of comparing NoSQL and PostgreSQL, we can observe that in general the mean time in NoSQL was lower than PostgreSQL, as was expected. But we have some exceptions like Shorts-Test with cache and geo-replication, all the User-Delete, User-Register without cache with geo-replication and Realistic-Flow without cache and with geo-replication. The User-Delete we attribute them the fact that PostgreSQL allow us to make SQL queries with DELETE, unlike NoSQL where we needed to first get the entity and then delete it making NoSQL in general slower then PostgreSQL. We had some issues with the PostgreSQL deletes, but we think that it does not interfere with these tests, because in the majority of the User-Delete we only had 1 or 2 errors with

code 500. The other exceptions were all in geo-replication that we believe could cause the problem, because in azure we checked and the PostgreSQL had only one region, but NoSQL had two, and we really think it could interfered with the results. Other reason that we think it can prove our idea is that, without geo-replication, we always got better values with NoSQL. Other reasons that could lead us to these exceptions are the fact that we have one container for all the entities stored in NoSQL. Because in some functions like getFeed (where we got a big difference on the mean response time values of each one: in PostgreSQL we got mean: 100,6; in NoSQL we got 160;) in PostgreSQL we can easily make a Inner Join, and with one query get all the information we want to return, but in NoSQL we need to do multiple queries for each of the containers, and perform many for cycles, that ends up getting worst processing time in the code.
In general we had the expected results, but some exceptions that could be based on the fact that we have no geo-replication in PostgreSQL, and with PostgreSQL we can perform queries on many tables at the same time, something that we can't do in NoSQL with the containers.

– **Analysis of using Redis Cache**

Analyzing our result on the tables on the purpose of comparing Cache vs NoCache, Realistic flow, and Upload Shorts got worst values with cache. We give that to the fact that we implemented a write through strategy, which turn the write functions much more slower, given the fact that we need to write on the DB and then write on the cache. That issue, lead us to worst mean values on the general of that tests. In User-Register we got always better values with cache, we give that to the fact that in those tests, many different gets of users are performed, leading us to get a fastest response reading them from the cache. During our implementation tests, we always got some unexpected values with cache, some times the gets were better, sometimes they were worse. If we just perform a succession of the same get, we got better values with no cache, like we can see in the User-Test. After analyzing and searching, we found that CosmosDB has an integrated cache, so then we tested that. When performing the first get from CosmosDB we got 300ms on postman, and then when performing a get from the cache we get 100ms. If we repeated the get from CosmosDB over and over again, we will always get values in the order of the 70ms, which are better then the 100ms that the cache gives us. That way we attribute this to the fact that we had worst values on our User-Test(where we repeat the same get some times), and probably could have influenced the Realistic flow tests. The most surprising values were the deletes, where we expected better values without cache, given the fact that, without it, on a delete operation we only remove items from the DB.
In sum we got the expected results, because during the implementation and debugs we noticed that we where implementing a really complex cache strategy. That probably turned it into a less efficient cache. But on the other hand

we ended up having better results as we where expecting.

- **Analysis of the impact of Geo-Replication**

  Using Geo-Replication has shown in the majority of the operations a great impact in response times compared to when not replicating the data (only showing a bit faster response times when using NoSQL with Cache). This impact was to be expected because the time of writing in 2 regions will be logically longer than when writing in only 1.

2 Conclusion

In conclusion, we consider we have done a good job in CosmosDB and Azure Blobs Storage, having had quite good results during tests. Related to cache implementation we were a bit greedy when trying to maintain a cache with all the information while trying to keep it up to date, leading us to a worse performance on these terms and the complexity size of our implementation also cost us time to implement the remaining features. After realizing cache's implementation was too complex, we decided to keep it because of the time we had left to finish the project's implementation and also because of the amount of work we put into it.