NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Department of Informatics

Master in Computer Science and Informatics

Azure TuKano

Project Assignment #1 - Azure PaaS

Student:
71750 - Rafael A. Costa

Professors:
Dr. Sérgio Duarte
Dr. Kevin Gallagher

November 2024

Conteúdo

1 Introduction

The goal of this project is to understand how the services available in cloud computing platforms can be used to create applications that are scalable, fast, and highly available.

This project consists of porting an existing web application to the Microsoft Azure Cloud platform. To that end, the centralized solution that was provided will need to be modified to leverage the Azure Platform as a Service (PaaS) portfolio, in ways that follow the current cloud computing engineering best practices.

To achieve the desired end result, we are expected to fulfill some requirements:

- Port the application to use Azure Blob Storage, Azure Cosmos DB and Azure Cache for Redis where deemed relevant.
- Implement a SQL and a NOSQL solution for the Azure Cosmos DB component.
- Leveraging azure services (in my case, azure functions) count the total views of a certain short and make a recommended account, based on user content.
- add Geo-Replication support

This report will discuss overall performance thought the solution for both the different database versions and for the cached/cacheless version. I will also discuss results for geo replicated scenarios with users across the globe

2 Blob Storage

The first requirement we set out to achieve was transferring shorts from the local storage to **Azure Blob Storage**. For that we had to implement a new version of BlobStorage (CloudStorage) where we store our blobs in the storage.

This part was rather simple, with the added caveat that on the Geo replicated solution, our blobs are uploaded to all of the storage accounts throughout the globe (in this case america and europe), but we only download from the nearest source.

We also elected to minimize the usage of the token feature through, as that was not a required implementations feature, and cause a lot of hard to solve behaviors, which weren't relevant on the current version of the solution, but we hope to tackle that problem in the upcoming second stage. minted

3 Data Base

3.1 Baseline

In this project there was already an implementation for a database in memory leveraging the jdbcDriver and a custom Hibernate solution. We were asked to implement both a the option to use a Azure SQL Database (in our case, PostgresSQL) and a Cosmos DB NoSQL.

3.2 SQL

we started with the SQL solution, as we were more familiarized with relational databases, and there already a hibernates file, and a presence of a lot of the desired configurations. All it took was adjusting the hibernates file, and the POM file to serve our purposes, and once properly configured for azure, it worked without much effort.

3.3 NoSQL

The Cosmos DB NoSql solution took a lot more work, as the projected was initially prepared for it. First we created a DB interface, that could abstract which db was being used, and created a DBFactory, that could decide based on startup configs which database to use. Then we created an implementation of this DB for our NoSql version.

This caused some major issues that we were able to circumvent efficiently, keeping the consumer code pretty much the same.

There was one exception: NoSql databases don't have the concept of transactions, but cosmos db does have something called batches. We used those batches to substitute the code for transactions from hibernate. but due to the way that transactions where implemented by default, the solution was less than optimal, having to force not only the consumer to change their code, but to have to implement a solution for each case. This is one of the main points we hope to fix for the upcoming second version, we didn't do it due to time constraints, and the lack of required technical know how.

3.4 Performance

3.4.1 Theoretical Performance

The performance of these models varies based on the operations. For reading operations, Relational databases tend to offers faster performance for complex queries, this is, data fetching that involves the aggregation of data available in multiple tables, due to the optimized query planners, indexing and the need the data being all on the same location. As for Non relational databases, they provided very efficient simple queries but have more delayed response times with complex queries due to the unstructured nature of the data, and the extra operations required to aggregate it.

For write operations, the roles reverse, due to the unstructured nature of data in a NoSQL environment, it tends to be inserted faster, wheres in a relational setting, the data needs be asserted as following all the criteria and placed in the correct locations so it takes longer.

3.5 Experimental evaluation

The comparison between **NoSQL** (Cosmos DB NoSQL) and **SQL** (Cosmos DB for PostgreSQL) without caching follows, it should be taken into account that this testing was done on a local environment, as to avoid unnecessary delays caused by access to the service itself, or container CPU over usage.

```
Sql, no cache
/rest/users/: (Create User)
  min: ..................................................... 236
  max: ..................................................... 4095
  mean: ..................................................... 326.7
  median: ..................................................... 237.5
  p95: ..................................................... 247.2
  p99: ..................................................... 3395.5
/rest/users/{{ id }}?pwd={{ pwd }}: (Get User)
  min: ..................................................... 312
  max: ..................................................... 356
  mean: ..................................................... 316.4
  median: ..................................................... 314.2
  p95: ..................................................... 320.6
  p99: ..................................................... 320.6


NoSql, no cache
/rest/users/: (Create User)
  min: ..................................................... 207
  max: ..................................................... 301
  mean: ..................................................... 218.2
  median: ..................................................... 210.6
  p95: ..................................................... 257.3
  p99: ..................................................... 290.1
/rest/users/{{ id }}?pwd={{ pwd }}: (Get User)
  min: ..................................................... 216
  max: ..................................................... 349
  mean: ..................................................... 227.2
  median: ..................................................... 219.2
  p95: ..................................................... 267.8
  p99: ..................................................... 327.1
```

3.6 Conclusion

At a first look, it might seem that our Cosmos DB for PostgreSQL might be completely outmatched, which looks weird, given our initial considerations. But in practice it makes perfect sense, as both the operations we used were simple read and writes, playing completely into the advantages of possessing a Cosmos DB NoSQL, so the results are to be expected. We'd do well to keep these results in mind, as we are going to bring them up shortly after, when we test them with a redis cache

4 Cache

4.1 Redis Cache

Implementing a Redis Cache was one of other objectives. For this we created on in azure, and locally created a class for it, we used with a JedisPool, for faster performance, so that we wouldn't have to initialize a Redis connection every time, and could already use a pre-existing one.

Using Redis on any project has some drawbacks and some upsides: Running a Redis cache is usually more expensive, and when misused can end up costing precious performance, due to the timing added of writing to a cache and checking it, in-case regular operations need to be done (If the desired item isn't present) BUT, it allows for much faster access when used in its intended way, generating a gigantic performance boost.

Due to this we elected to only use Redis for creating, deleting and searching for users and shorts, as we deemed those as the operations that would be occurring the most often repeatedly, maximizing the usage of our cache, while minimizing the downsides.

A interesting point is that we choose to NOT add searches to our cache, as that would only work if the user did the exact same database search, if that were not the case we would have to access the database anyway, defeating the point of having a cache in the middle.

4.1.1 Theoretical Performance

Due to the way we know caches work, we expect the response time on cached data to be much lower, compared to not cached instances, as that is the purpose of cache, and given

4.2 Experimental evaluation

```
Sql, cache
/rest/users/: (Create User)
  min: ..................................................... 78
  max: ..................................................... 109
  mean: ..................................................... 80.1
  median: ..................................................... 80.6
  p95: ..................................................... 82.3
  p99: ..................................................... 85.6
/rest/users/{{ id }}?pwd={{ pwd }}: (Get User)
  min: ..................................................... 192
  max: ..................................................... 256
  mean: ..................................................... 195.9
  median: ..................................................... 194.4
  p95: ..................................................... 198.4
  p99: ..................................................... 214.9
```

```
NoSql, cache
/rest/users/: (Create User)
  min: ......................................... 51
  max: ......................................... 123
  mean: ......................................... 68.1
  median: ......................................... 53
  p95: ......................................... 106.7
  p99: ......................................... 117.9
/rest/users/{{ id }}?pwd={{ pwd }}: (Get User)
  min: ......................................... 98
  max: ......................................... 223
  mean: ......................................... 121.1
  median: ......................................... 102.5
  p95: ......................................... 169
  p99: ......................................... 214.9
```

Comparison between cached and not cached version clearly tells us that cache data is faster to get, which is no surprise, and at an average of 100 or more ms! The usage of cache should be expaneded and furthered pondered on future stages of work, if we see ourselves in a situation where our app calls a lot of the same API endpoints, in a easy to replicate way.

5 Azure-Functions

5.1 Counting Views

One of the Azure Functions implemented in our project is responsible for counting the views of the shorts. Every time a video blob is downloaded, Azure Blob Storage emits an event, which then triggers our function. Once triggered, the function retrieves the corresponding Short from Cosmos DB, increments its totalViews field, and writes the updated record back to the database.

Or that would be the idea. In practice, azure doesn't allow for a trigger to be generated via a blob download, which completely destroys this approach. We explored another avenue, using a HTTP function that is called when a blob is downloaded, but that is just a convoluted approach and felt like forcing the issue, so to say, as at that point why not just launch a new thread and do the request server side? There are much better approaches. We settled for doing just that, just not with the extra thread, as we hope to find a more correct solutions come second half of this project

The implementation of this function allows us to handle the view counting logic completely server-side, without affecting the user's experience. Every time a user accesses a short, the view is logged in the background, ensuring that our system remains performant and responsive. If it worked.

5.2 TuKano Recommends

The second Azure Function implemented is responsible for managing the content of "TuKano Recommends,"a system user designed to promote the most popular shorts on the platform. This function is triggered periodically using a timer and queries the top 10 most viewed shorts from Cosmos DB.

Once the top shorts are identified, the function republishes them under the 'TuKano Recommends' user. This ensures that all users of the platform can access a curated list of the most popular content. Before running its main logic, the function also checks if the system user exists in the database and creates it if it doesn't.

This is a kinda perfect fit for azure functions, as it avoids colision with regular operations, and is a rather costly one.

5.3 Conclusion

Even though we failed our goal with the counting views implementation, we have every intention to bide our time, and wait for the second half of the project to come around and fix these issues!

6 Geo-Replication

6.1 Setup

In this project, we implemented geo-replication to enhance availability and fault tolerance by deploying resources across multiple regions. Specifically, we set up a secondary blob storage in centralus to work alongside the primary storage in euwest. Additionally, our NoSql Cosmos Database instance was configured with geo-replication, ensuring data synchronization between regions. To complete the setup, we deployed a web application in the eastus2 region, configuring it to use the replicated blob and database resources as its primary endpoints.

6.2 Blob Storage Replication

To enable geo-replication for blob storage, the ‘CloudStorage‘ class was extended to handle multiple storage accounts. The solution leverages two ‘BlobContainerClient‘ instances: one for the primary blob storage in NA and another for the secondary storage in EU. This dual setup ensures that data is always available and up-to-date across regions.

For write operations, the ‘write‘ method uploads the blob to both the NA and EU storage accounts. This guarantees that data in both regions is synchronized immediately after any write operation. For read operations, the implementation prioritizes the NA storage, which serves as the primary region. If the blob is not found in NA, the method falls back to the EU storage. This ensures data availability even if the primary region experiences downtime.

6.3 NoSQL Database Replication

For the database layer, we used Cosmos DB NoSQL, which supports automatic geo-replication. This feature ensures that any data written in one region is automatically replicated to other regions. By enabling geo-replication between the NA and EU regions, the system ensures data consistency without requiring manual intervention.

Unlike the blob storage replication, this setup required minimal changes to the application code. Cosmos DB handles all replication tasks transparently, making it an ideal choice for projects that require robust multi-region support without the overhead of complex configurations. This setup allows our application in NA to operate independently of the EU database while keeping all data in sync across regions.

6.4 Why No PostgreSQL?

Initially, we considered implementing a similar geo-replication strategy for our PostgreSQL database using Cosmos DB for PostgreSQL. However, after evaluating the costs associated with enabling geo-replication for PostgreSQL, it became clear that this option was not financially viable. The additional costs were significant, and given our project's constraints, it made more sense to focus on the NoSQL solution, which offered comparable performance and scalability at a fraction of the cost.

Additionally, the NoSQL database met all our operational requirements, making the added expense of PostgreSQL replication unnecessary. This decision allowed us to stay within budget while still providing a robust and efficient data layer for our application.

6.5 Web Application Deployment

Given that this is meant to work with the intent of world wide users, it made a lot of sense to deploy our app directly to the us aswell, and that is what we did. Of course we made it leverage our storage solutions already there

6.6 Experimental evaluation

We found it fitting to also test what the performance would be like if, per example, the eu platform was down. Here is the data!

```
NoSql, no cache, EU (repeated data)
/rest/users/: (Create User)
  min: ...................................................... 207
  max: ...................................................... 301
  mean: ..................................................... 218.2
  median: ..................................................... 210.6
  p95: ...................................................... 257.3
  p99: ...................................................... 290.1
/rest/users/{{ id }}?pwd={{ pwd }}: (Get User)
  min: ...................................................... 216
  max: ...................................................... 349
  mean: ..................................................... 227.2
  median: ..................................................... 219.2
  p95: ...................................................... 267.8
  p99: ...................................................... 327.1

NoSql, no cache, NA
/rest/users/: (Create User)
  min: ...................................................... 380
  max: ...................................................... 1977
  mean: ..................................................... 441.7
  median: ..................................................... 391.6
  p95: ...................................................... 699.4
  p99: ...................................................... 1085.9
/rest/users/{{ id }}?pwd={{ pwd }}: (Get User)
  min: ...................................................... 382
  max: ...................................................... 1054
  mean: ..................................................... 410.4
  median: ..................................................... 391.6
  p95: ...................................................... 518.1
  p99: ...................................................... 820.7
```

6.7 Conclusion

Surprising no one, it would be much slower to use the app like this! The good news is that it really isn't that bad (a mere 200 ms extra on average is manageable). This is a good signal that this type of deployment would help sustain the application and its working and running, in case of critical region failure. Additionally, it would allow for users in other regions to not suffer from that same 200ms extra, which is a great help on user happiness and app performance.

7 Conclusion

In conclusion, we were able to (almost) meet all the demands of what was asked of us for this project, and what we failed, we learned and are expecting to make a great return for second part! We have a good implementation of: A Cosmos DB NoSql, a Cosmos PostgreSQL database, a Redis Cache, one solid azure function, Geo-Replication and an App service. We hope to improve code visibility and interchangeability of options (forgot to add ENV variables...) at a later stage, as well as tackling the issues were we failed. We hope to get corrected on the larger issues to improve!