Department of Informatics

Master in Computer Science and Informatics

Azure TuKano

Project Assignment #1 - Azure PaaS

Students:
70730 - Rodrigo H. Correia
70731 - Carlos G. Pereira

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.

As part of the end result, this report will explain the design choices and provide a performance evaluation of the solution for the original application and the one deployed to the Azure Cloud platform.

2 Storage

2.1 Blob Storage

One of the initial requirements was to convert the way shorts were stored. In the provided application, this was done by leveraging the local file System. Now, with our improvements, the application is taking advantage of **Azure Blob Storage**, which is a more reliable and scalable solution.

The application also leverages geo-replication for blob storage, by uploading the blobs to all regions but only downloading from the region where the application is running. This way, we keep the data consistent across all application instances whilst allowing for low latency for retrieval of the data. minted

3 Data Base

3.1 SQL vs NoSQL

In this project, two database implementations were evaluated: **Cosmos DB for PostgreSQL** for SQL and **Cosmos DB NoSQL** for NoSQL.

The main difference between the two implementations is their approaches to storing the data. **SQL** relies on a relational model, with structured tables and relations between them. While **NoSQL** uses a more flexible model, such as document-based storage in JSON-like formats, with no needed relation or structure inside them. This allows for dynamic and hierarchical data structures without the need for predefined schemas.

3.1.1 Performance

The performance of these models varies based on the operations. For reading operations, **Postgres** 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 and indexing. As for **NoSQL**, it provides very efficient simple queries but has more delayed response times with complex queries. For write

operations, **Postgres** has a rigid insertion because of the schema that needs to be strictly followed, while **NoSQL** excels in high-speed data insertions because of the unstructured approach.

3.1.2 Scalability and Flexibility

- **SQL Databases**: Favors vertical scaling by enhancing server resources, like CPUs and RAM size, but **Cosmos DB for PostgreSQL** also supports horizontal scaling. These databases are very easy to ensure data integrity and consistency, which is vital for transactional applications that require all-round consistency.
- **NoSQL Databases**: Very easy to scale horizontally, by just increasing the amount of servers to handle large data volumes and high traffic.

3.2 Experimental evaluation

The comparison between **NoSQL** (Cosmos DB NoSQL) and **SQL** (Cosmos DB for PostgreSQL) without caching, highlights significant performance differences. The NoSQL implementation, demonstrates faster response times, with median response times ranging from 90.9 ms to 228 ms, as seen below:

```
http.response_time:
  median: ..................................................... 183
```

In contrast, the SQL implementation shows median response times ranging from 267.8 ms to 632.8 ms, as indicated in:

```
http.response_time:
  median: ..................................................... 267.8
```

For higher percentiles, NoSQL maintains a P95 of 368.8 ms and a P99 of 459.5 ms:

```
http.response_time:
  p95: ........................................................ 368.8
  p99: ........................................................ 459.5
```

While SQL shows significantly worse performance with a P95 of 804.5 ms and a P99 of 837.3 ms:

```
http.response_time:
  p95: ........................................................ 804.5
  p99: ........................................................ 837.3
```

These results indicate that NoSQL outperforms SQL in terms of latency and efficiency under similar load conditions, making it better suited for high-throughput, low-latency operations for simple queries like the one used to obtain these results.

4 Cache

4.1 Redis Cache

By implementing a Redis Cache service it was possible to greatly decrease the amount of database accesses. This occurs because in our application, we check if the requested data is in the cache, and only if it is not, do we send a request to the databases to retrieve it. Afterward this search, and as a strategy to decrease future accesses, we store the results in the cache. This cache implementation, despite being very powerful in providing very quick response times by minimizing database accesses that can be slow, must be used with some caution to provide the most up-to-date answers to our users, because we might be giving our users some information, that if poorly cached, might be outdated.

For example, in our project we don't use cache for every endpoint, we don't use this for Blob-related operations, because this would be a lot of information to cache since our blobs are the content of our shorts and this would make no sense. Another example where the cache is not used is in the likes of our shorts because the number we are getting might be outdated.

A proper cache usage in our project is for example when a client tries to create a user, before sending the request to the database with that intent and then maybe getting a CONFLICT response, we check immediately if the user that is trying to be created is already cached in the list of available users if it is, we can correctly affirm that this is a CONFLICT without needing a database connection.

Initially, in our development process, we were thinking of implementing a cache on the search users' operation, but upon further discussions, we thought that for a search in the database, to ALWAYS provide accurate results of the users that match the search criteria, we would need to get them from the database, therefore eliminating the need to try to cache search results.

4.2 Experimental evaluation

The comparison between the NoSQL implementation with and without caching also reveals expected, yet significant performance improvements when caching is enabled. Without caching, median response times range from 183 ms to 228 ms, as seen in:

```
http.response_time:
  median: ..................................................... 183
```

However, with caching, these values drop significantly, with median response times ranging from 53 ms to 102.5 ms, seen in:

```
http.response_time:
  median: ..................................................... 53
```

At higher percentiles, the P95 and P99 response times show a drastic reduction with caching. Without caching, P95 values peak at 368.8 ms and P99 at 459.5 ms. With caching, these values are reduced to 314.2 ms (P95) and 399.5 ms (P99), as seen in:

```
http.response_time:
  p95: ..................................................... 314.2
  p99: ..................................................... 399.5
```

These results demonstrate that caching significantly enhances performance, reducing response times and variability across all metrics, making the system far more responsive and consistent under load.

5 Conclusion

In summary, we were able to convert the application to correctly utilize Azure Cloud Platform functionalities, including Blob Storage for storing big data, Redis Cache to reduce the number of accesses to the Cosmos DB, which kept a record of the current users, their information, and respective shorts. We also provide multiple configurations, for different purposes and needs, including no cache vs cache, no SQL vs SQL, all with the use of different environment variables to pick between them. This is not only good for the customer, but also for development and testing purposes, because it is possible to test individual components.

Taking into consideration the current state of the software, the application can be easily deployed and replicated with all the mandatory requirements successfully.

The performance results can be observed on the benchmark test presented on the git repository.