SCC Project 1

Rodrigo Silva Nº70567
António Salselas Nº70568

10th November 2024

1 Introduction

In today's digital landscape, cloud computing has become essential for developing applications that are scalable, fast, and highly available. By leveraging cloud services, developers can build solutions that meet the evolving demands of users while reducing infrastructure and operational overhead. This project aims to explore the benefits of cloud computing by deploying an application built from a foundational codebase, named TuKano, to the Microsoft Azure platform.

The project involves adapting TuKano, a base code provided by the professors, to function within Azure's Platform as a Service (PaaS) environment. This adaptation will transform TuKano by incorporating Azure's managed services—such as app hosting, database solutions, and container management-following best practices in cloud engineering. The goal is to create a cloud-native application that demonstrates enhanced performance, resilience, and scalability.

This report details the migration process, key design considerations, and implementation strategies used to align TuKano with Azure's cloud-native capabilities. Through this project, we aim to highlight the steps and challenges of building applications that leverage the full potential of a modern cloud platform.

2 Ported Solution Description

TuKano is structured as a three-tier architecture, with the application tier consisting of three REST services: Blobs, Users, and Shorts. While migrating the TuKano application to Azure's Platform-as-a-Service (PaaS) environment, we made several decisions about which components to utilize. Ultimately, we settled on the following components:

- **Azure Blob Storage**: We selected Azure Blob Storage to store the multimedia content for shorts, as it provides a scalable, cost-effective solution for storing, managing, and delivering large amounts of media. Azure Blob Storage is optimized for handling multimedia files, allowing us to efficiently serve high-quality content to users without compromising performance or incurring high storage costs. This choice enables a smoother, more reliable user experience, as it supports rapid content retrieval and streaming, even during periods of high demand. To integrate Azure Blob Storage into our application, we modified the FilesystemStorage class to connect directly to the designated Azure container, replacing the default local storage setup. This adjustment ensures that all multimedia content is stored in the cloud, where it can be accessed and managed centrally. By configuring FilesystemStorage to interact with Azure's container, we enable seamless scalability and take advantage of Azure's built-in features, such as redundancy, security, and accessibility, which enhance data integrity and make it easier to maintain consistent performance across our application.

- **Azure CosmosDB**: This resource is well-suited for managing individual user data, shorts metadata, and key social networking features such as user feeds, follower relationships, and likes. To accommodate different data management needs, we implemented two solutions: CosmosDB NoSQL and CosmosDB for PostgreSQL. The primary distinction between these solutions lies in their query mechanisms. For CosmosDB NoSQL, we leveraged its native API, which allows us to work with flexible, schema-free data structures ideal for scalable social features. For CosmosDB for PostgreSQL, we opted to use Hibernate for object-relational mapping, enabling efficient relational data queries. To integrate both resources seamlessly, we adapted the query syntax for each database and introduced a configuration flag to specify which resource is being accessed at runtime. Each database has its own dedicated class within the DB folder, encapsulating the necessary code and configurations for easy maintenance and scalability. This setup provides flexibility, allowing us to switch between the two resources based on specific needs or performance requirements.

- **Azure Cache for Redis**: By integrating Azure Cache for Redis, TuKano benefits from significantly faster data retrieval, enhanced scalability, and improved overall performance. This caching layer allows us to quickly serve frequently accessed data, reducing load times and minimizing the need to repeatedly query the primary database. However, during the performance testing provided by the professors using Artillery, we identified a small error in our Redis implementation. This error caused the application to crash, requiring a new deployment to resolve it. As a temporary measure, we limited the test size to maintain functionality without interruptions. Our Redis integration is organized into two main classes: one dedicated to initializing the Azure Cache for Redis instance, and another to manage the key operations—such as set, get, and delete—needed to interact with cached data. In our implementation, caching is used selectively. We focus on caching user and short creation and retrieval operations, which are frequent and benefit most from the performance boost. Other operations, like managing user feeds or deleting all shorts, are excluded from caching because they involve more complex data processing or require fetching older records from the database. For a high-traffic application with thousands of users and shorts, directly querying the database for these less common actions helps ensure data accuracy and consistency.
- **Azure Functions**: This resource was introduced in the final phase of the project to create a system that manages the incrementing of video views. While the implementation is complete, a minor error prevented us from fully testing it. This issue is likely related to how the function is invoked within the JavaBlobs class. Despite this, we have included the code with our implementation for reference.

3 Performance Evaluation

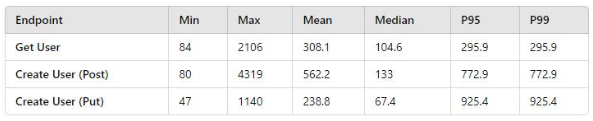
Once all the necessary alterations and initial tests were completed, we moved on to conducting performance tests to evaluate the system's efficiency under various conditions. In these tests, we compared the application's performance with and without the use of caching, specifically utilizing Azure Cache for Redis. We also tested the two database alternatives—CosmosDB NoSQL and CosmosDB for PostgreSQL with Hibernate. The goal was to assess how the caching layer impacted data retrieval times, as well as to analyze the performance differences between CosmosDB NoSQL and CosmosDB for PostgreSQL. The following subsections reflect our findings.

3.1 With Cache Vs Without Cache

In this scenario, we ran the application both with and without Azure Cache for Redis to evaluate its impact on performance. We selected CosmosDB NoSQL as the resource for database management in this case. The performance tests focused on comparing the application's behavior and response times under two different conditions: one with caching enabled and the other without.

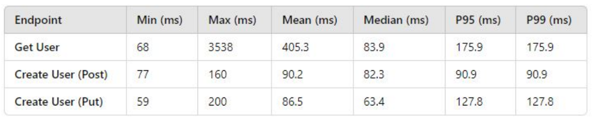
We specifically measured the performance during key operations, such as creating and retrieving a user, as well as retrieving and uploading a short. These actions were chosen because they are among the most frequent in the application, and their performance is crucial for overall user experience.

The following images illustrate the performance differences between the two configurations. They show the time taken to create and fetch a user, retrieve a short, and upload a new one, highlighting how the use of Azure Cache for Redis influences response times and system efficiency.



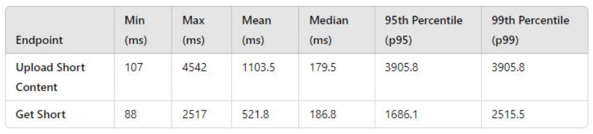
Endpoint	Min	Max	Mean	Median	P95	P99
Get User	84	2106	308.1	104.6	295.9	295.9
Create User (Post)	80	4319	562.2	133	772.9	772.9
Create User (Put)	47	1140	238.8	67.4	925.4	925.4

Figure 1: With Cache - User



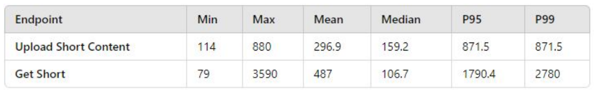
Endpoint	Min (ms)	Max (ms)	Mean (ms)	Median (ms)	P95 (ms)	P99 (ms)
Get User	68	3538	405.3	83.9	175.9	175.9
Create User (Post)	77	160	90.2	82.3	90.9	90.9
Create User (Put)	59	200	86.5	63.4	127.8	127.8

Figure 2: Without Cache - User



Endpoint	Min (ms)	Max (ms)	Mean (ms)	Median (ms)	95th Percentile (p95)	99th Percentile (p99)
Upload Short Content	107	4542	1103.5	179.5	3905.8	3905.8
Get Short	88	2517	521.8	186.8	1686.1	2515.5

Figure 3: With Cache - Short



Endpoint	Min	Max	Mean	Median	P95	P99
Upload Short Content	114	880	296.9	159.2	871.5	871.5
Get Short	79	3590	487	106.7	1790.4	2780

Figure 4: Without Cache - Short

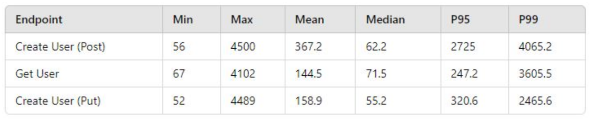
We can conclude that for the "Create" operation, the version without cache was faster, as there was no need to write the value to the cache, only to the database itself. Regarding the "Get" operation, the test with cache performed slower than the version without, likely due to the small test size. This performance discrepancy is attributed to a minor error in the cache implementation, which prevents the use of larger test samples. It is important to note that these results are not entirely accurate due to the limited sample size used in the tests.

3.2 CosmosDB NoSQL vs CosmosDB for PostgreSQL with Hibernate

In this scenario, we ran the application using either CosmosDB NoSQL or CosmosDB for PostgreSQL with Hibernate, and compared the performance results between the two configurations. To ensure the most accurate and reliable results, we decided to exclude Azure Cache for Redis from this particular test. This allowed us to focus solely on the performance of the databases themselves, without the influence of caching, which could potentially skew the results.

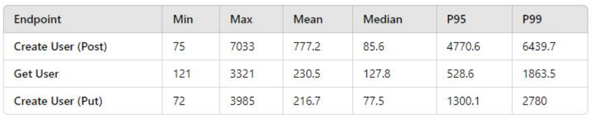
The performance was measured for several key operations, including creating and retrieving a user, as well as retrieving and uploading a short. These actions were chosen because they represent common tasks that users frequently perform in the application.

The following images illustrate the performance differences observed between the two database solutions, highlighting the variations in response times for each operation. These results provide insights into how CosmosDB NoSQL and CosmosDB for PostgreSQL with Hibernate perform under similar conditions, offering a clearer picture of which database solution might be more suitable.



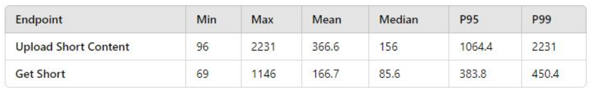
Endpoint	Min	Max	Mean	Median	P95	P99
Create User (Post)	56	4500	367.2	62.2	2725	4065.2
Get User	67	4102	144.5	71.5	247.2	3605.5
Create User (Put)	52	4489	158.9	55.2	320.6	2465.6

Figure 5: With NoSQL - User



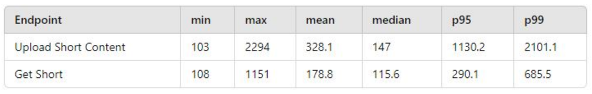
Endpoint	Min	Max	Mean	Median	P95	P99
Create User (Post)	75	7033	777.2	85.6	4770.6	6439.7
Get User	121	3321	230.5	127.8	528.6	1863.5
Create User (Put)	72	3985	216.7	77.5	1300.1	2780

Figure 6: With PostgreSQL - User



Endpoint	Min	Max	Mean	Median	P95	P99
Upload Short Content	96	2231	366.6	156	1064.4	2231
Get Short	69	1146	166.7	85.6	383.8	450.4

Figure 7: With NoSQL - Short



Endpoint	min	max	mean	median	p95	p99
Upload Short Content	103	2294	328.1	147	1130.2	2101.1
Get Short	108	1151	178.8	115.6	290.1	685.5

Figure 8: With PostgreSQL - Short

In conclusion, NoSQL databases outperform PostgreSQL in terms of speed. NoSQL systems are designed to handle large volumes of unstructured data with high scalability, which enables them to process requests much faster than relational databases like PostgreSQL. Their flexibility in schema design and ability to distribute data across multiple nodes also contribute to superior performance, especially in applications that require rapid data retrieval and real-time processing. On the other hand, while PostgreSQL excels in complex queries and maintaining strong data consistency, it tends to be slower when handling high-throughput workloads or large-scale, distributed data scenarios compared to NoSQL systems.

4 Conclusion

In conclusion, this project demonstrated the significant benefits of leveraging cloud-native technologies to enhance the performance, scalability, and resilience of applications. By adapting the TuKano codebase to the Microsoft Azure platform, we utilized key Azure services such as Blob Storage, CosmosDB, Azure Cache for Redis, and Azure Functions to optimize various aspects of the application.

The integration of Azure Blob Storage for multimedia content storage, CosmosDB for efficient data management, and Azure Cache for Redis for faster data retrieval all contributed to improving the overall performance and user experience. Despite encountering some challenges, such as a minor error in the Redis implementation and limitations in the testing environment, we were able to identify areas where caching and database selection play crucial roles in enhancing response times.

The performance evaluation indicated that, in general, caching significantly improves the efficiency of data retrieval, although it was not always effective under small test conditions due to the Redis error. Moreover, the comparison between CosmosDB NoSQL and CosmosDB for PostgreSQL with Hibernate showed that database choice impacts performance.

Overall, this project highlights the value of adopting cloud services and best practices to build scalable, high-performance applications. The lessons learned from this implementation will guide future cloud-based projects, showcasing the power of cloud platforms in meeting modern application demands.