Deploying Tukano in the Cloud

Cloud project development

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

José Romano 59241

Pedro Cavaleiro 57974

Computational systems and cloud

2024/2025

1. Introduction

The objective of this project is to explore and demonstrate the advantages of cloud-based services by migrating the existing TuKano application to the Microsoft Azure Cloud ecosystem. By leveraging Azure's Platform-as-a-Service (PaaS) offerings, our goal is to enhance the application's scalability, availability, and performance within a cloud environment.

In this transition, we focused on optimizing the data layer of the application by integrating cloud-native data services. Specifically, we utilized Azure Blob Storage for managing and storing video files, Azure Cosmos DB with PostgreSQL for structured data management, and Azure Cache for Redis to improve performance by reducing latency and enhancing data retrieval speeds.

Throughout this report, we will discuss the architecture changes, design decisions, and performance analyses that illustrate the impact of our cloud migration approach on the TuKano application. This exploration aims to highlight both the technical challenges encountered and the performance improvements achieved by leveraging the Azure Cloud platform.

2. Architecture and Design Decisions

In this section, we detail the changes made to TuKano's architecture for the migration to the Azure Cloud environment, highlighting the specific design decisions for each service and component of the data layer. The aim was to optimize data storage, caching and persistence to improve the application's scalability and performance.

2.1 Storage of Blobs

To store the videos (blobs) of the TuKano application, we used Azure Blob Storage. This Azure service is designed to store large volumes of unstructured data, such as documents, image, audio and video files. Blob Storage organizes data into "containers", which allow logical separation of sets of files and facilitate efficient access to the stored data.

The term "blob" (Binary Large Object) refers to the storage of large volumes of binary data. This feature makes Azure Blob Storage particularly suitable for applications that need to manage large files, such as videos, as it offers:

- **Scalability**: Blob Storage can scale automatically to accommodate a high volume of data, supporting intensive access scenarios such as those expected at TuKano.

- **High Availability and Durability:** With multiple redundancy options, Blob Storage ensures that data is highly available and protected against failures.

- **(Future) Integration with Content Delivery Networks (CDNs):** This feature allows content, such as videos, to be distributed to users quickly and with low latency, an essential feature for streaming applications or social networks with a strong video component, such as TuKano.

Given TuKano's need to provide an agile video viewing experience, Azure Blob Storage provides the ideal combination of storage efficiency and file retrieval speed, ensuring that the application is scalable and suitable for a high volume of uploads and views.

2.2 Cache for Blobs : Redis Cache

Azure Cache for Redis was chosen to cache blobs (videos) in order to optimize the retrieval performance of popular videos. In applications such as TuKano, where media content is highly requested, the cache acts as a high-speed in-memory storage layer that temporarily stores the most accessed data. This strategy reduces latency in read operations, especially in scenarios with a high volume of requests, where repeated requests can generate high loads on the storage layer.

Benefits of Redis Cache in Blobs:

- **Low Latency:** By keeping frequently accessed videos in memory, Redis provides faster data retrieval than traditional storage. This improves the user experience, especially for high-demand content.

- **Reduced load on Blob Storage:** By reducing repeated accesses to Blob Storage, the cache helps to reduce operating costs and optimize the use of resources, an advantage for applications that expect a high volume of simultaneous accesses.

- **Scalability:** Redis allows you to scale horizontally, supporting large read loads and ensuring that performance remains stable as the user base grows.

Why caching was applied only to blobs:

We chose to implement caching **exclusively on blobs** because we believed that, in the context of TuKano, **videos would be accessed much more frequently than other data**, such as user information. Caching user information, although technically possible, would be less efficient, as the access profile would not justify the intensive use of memory. Even if Redis manages the most requested data intelligently, including user information would consume unnecessary resources, since the performance gain would be negligible compared to caching videos.

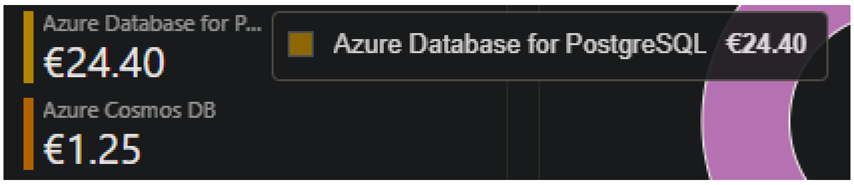
2.3 Data Structures of Users and Shorts: Cosmos DB

To store user data and video metadata (shorts), we implemented two configurations in Azure Cosmos DB: a SQL version and a NoSQL version. This approach was intended to enable a comparative analysis between the two, exploring the differences in terms of costs and scalability, and assessing which of the approaches might be more advantageous for the performance and management of the data in the TuKano application.

Motivation for using Cosmos DB:

- **Flexibility for Complex Queries:** Both the SQL and NoSQL versions of Cosmos DB offer support for efficient and scalable queries, which are fundamental for fast and well-defined operations on user and metadata data.

- **Adaptation to Cost Variations and Scalability:** One of the advantages of Cosmos DB is that it allows you to migrate between SQL and NoSQL as the application's needs evolve, especially in growth scenarios. The NoSQL option, for example, is generally more scalable and can reduce costs in certain situations, while the SQL version may be more suitable for data with structured relationships.


Azure Database for P...
€24.40
Azure Cosmos DB
€1.25
Azure Database for PostgreSQL €24.40

(The Figure above illustrates the cost difference between PostgreSQL and CosmosDB according to Azure cost estimation).

Note: On this commit point, we had PostgreSQL working. After that, for design reasons, we focused on CosmosDB and PostgreSQL stopped working as it was

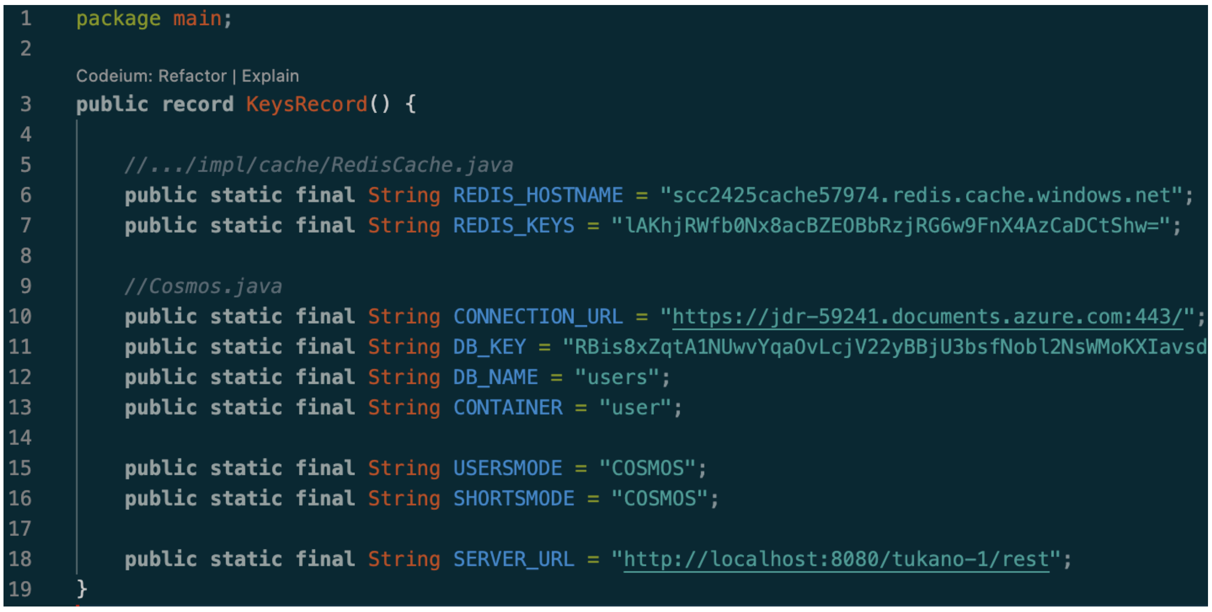
Why Blob Storage is not suitable for this: Azure Blob Storage is designed to store large binary files (blobs), such as videos or images, with no metadata structure or support for complex queries. While it is an excellent choice for storing media files (such as videos), it is not suitable for relational data or for queries that require filters and indexes, such as in metadata management.

3. Implementation

3.1 *KeysRecord*

A fundamental part of the implementation was the creation of KeysRecord.java, a Java record file whose main responsibility is to centralize the keys and credentials for accessing the resources we use on the Azure platform. This approach provides greater organization as it allows all critical keys and parameters to be configured at once, avoiding scattered configurations in different parts of the code.

KeysRecord.java stores the access keys and URLs needed to interact with Azure services such as Redis Cache and Cosmos DB, as well as the configuration of the application's server address. It also makes it easier to maintain the code, as any changes to the keys or settings only need to be made in this central file, without the need to search for all the places where this information is used.


```java
package main;

Codeium: Refactor | Explain
public record KeysRecord() {

    //.../impl/cache/RedisCache.java
    public static final String REDIS_HOSTNAME = "scc2425cache57974.redis.cache.windows.net";
    public static final String REDIS_KEYS = "lAKhjRWfb0Nx8acBZEOBbRzjRG6w9FnX4AzCaDCtShw=";

    //Cosmos.java
    public static final String CONNECTION_URL = "https://jdr-59241.documents.azure.com:443/";
    public static final String DB_KEY = "RBis8xZqtA1NUwvYqaOvLcjV22yBBjU3bsfNobl2NsWMoKXIavsd
    public static final String DB_NAME = "users";
    public static final String CONTAINER = "user";

    public static final String USERSMODE = "COSMOS";
    public static final String SHORTSMODE = "COSMOS";

    public static final String SERVER_URL = "http://localhost:8080/tukano-1/rest";
}
```

Improvements: It would be better if we used a *.props* file. This approach improves security, as we can hide credentials and sensitive information, as well as making the application more flexible by supporting different configurations for development, test and production environments.

3.2 userId -> id

During the implementation, we encountered conflicts due to inconsistent naming of the argument used to represent the user identifier. Initially, we were using userId, but this led to version conflicts and confusion in the codebase. To resolve this issue and ensure consistency across the project, we standardized the naming to id. This change streamlined the code and reduced potential errors caused by the mismatch of variable names across different parts of the application.

By renaming userId to id, we also ensured that the identifier is more general and can be used in a wider context, improving the clarity and maintainability of the code.

3.3 Hibernate compatible with Cosmos

To enable compatibility between Hibernate and Azure Cosmos DB, we configured the Hibernate settings within the hibernate.cfg.xml file. This configuration ensures that our application can connect to the PostgreSQL-compatible endpoint provided by Cosmos DB, allowing us to leverage the features of Hibernate for object-relational mapping (ORM) while using Cosmos DB as the underlying data store.

3.4 Adjustment of the Log System

We've added a more robust logging system to monitor and diagnose integration with Azure services. The logging library has been configured to output more detailed debug information about the connection to the database and the CRUD processes performed.

<u>Logging changes:</u>

- Use of the Logger class to record operations performed in various functions such as user creation, deleting and editing.

- Logs are now more informative, which makes it easier to diagnose possible problems during integration with the Azure infrastructure.

Result: This improved the visibility of the operations carried out in the application and provided greater control over the processes in production, helping with error detection and system performance.

3.5 JavaUsers, JavaShorts & JavaBlobs

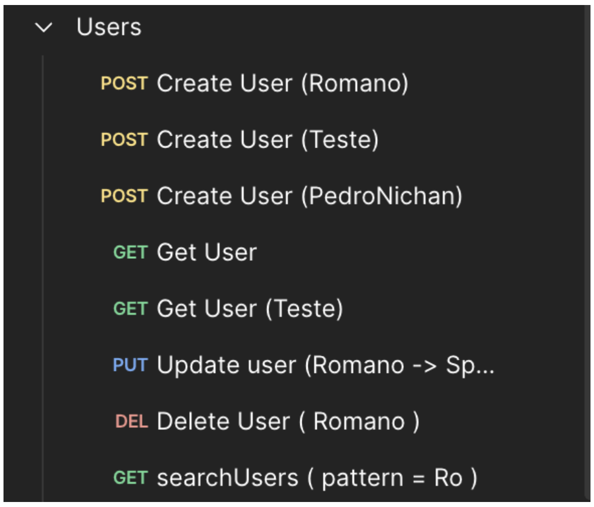
The CosmosDB implementation for these three classes was inspired by lab3, where we learned how to integrate CosmosDB efficiently and correctly into a system. This approach helped us to structure access to the database and ensure that the operations of creating, updating and deleting users, shorts and blobs were carried out correctly, taking advantage of the benefits of Azure CosmosDB.

4. Development tests

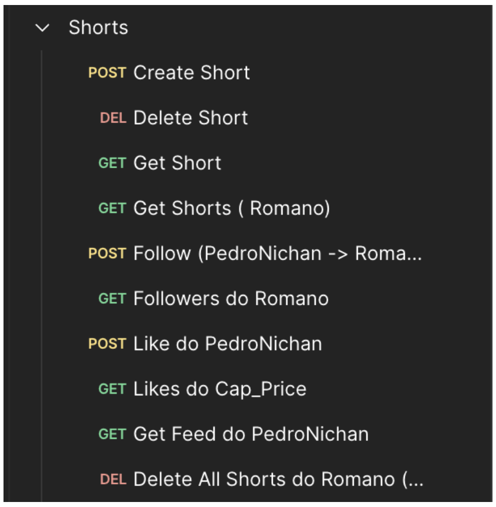
To validate that our application was working correctly, we carried out a series of tests using Postman. Initially, the tests were carried out with the local Tomcat server, ensuring that communication with the cloud resources was working correctly. We then repeated the tests using the Azure App Service, and the results confirmed that the application continued to work as expected, maintaining integration with the cloud services.

The operations tested involved the main flows of the application, including uploading and downloading blobs, as well as verifying the correct storage and retrieval of data in Azure Blob Storage and Redis Cache. All the tests returned a 200 OK code, indicating that the operations were being performed correctly. Although we didn't carry out specific tests to validate the Redis cache, we did observe an increase in writes, indicating that storage operations were being carried out successfully.

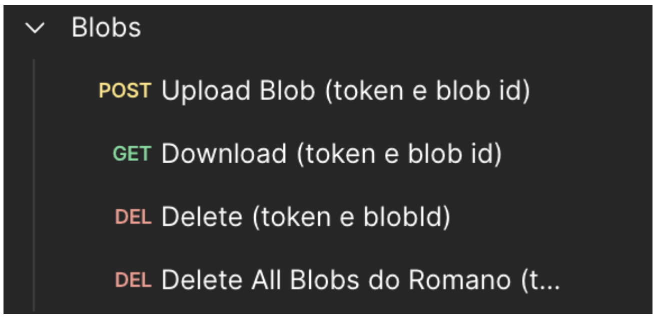
(The figures below show what tests we did for the Users, Shorts and Blobs and also a link in case you want to join our Postman Workspace.)



Users
POST Create User (Romano)
POST Create User (Teste)
POST Create User (PedroNichan)
GET Get User
GET Get User (Teste)
PUT Update user (Romano -> Sp...
DEL Delete User (Romano)
GET searchUsers (pattern = Ro)



Shorts
POST Create Short
DEL Delete Short
GET Get Short
GET Get Shorts (Romano)
POST Follow (PedroNichan -> Roma...
GET Followers do Romano
POST Like do PedroNichan
GET Likes do Cap_Price
GET Get Feed do PedroNichan
DEL Delete All Shorts do Romano (...



Blobs
POST Upload Blob (token e blob id)
GET Download (token e blob id)
DEL Delete (token e blobId)
DEL Delete All Blobs do Romano (t...

5. Discussion

During the development of the project, we faced initial challenges in debugging due to the absence of a logging system. Without clear logs, it was difficult to trace and precisely identify the causes of system failures, which made troubleshooting difficult and compromised development efficiency. After implementing the log system, the process became significantly more fluid, providing detailed information about the execution flow and errors, which facilitated both problem analysis and code improvement.

Furthermore, using Azure brought some additional difficulties. Deployment times were sometimes long, generating waiting periods and interrupting the workflow. At times, the temporary inactivity of resources on the platform also limited access to services and delayed system checks and tests. Even with these limitations, the use of Azure was essential to exploit the application's potential in the cloud.

We started the project with the “left foot” since we initially implemented some changes to other components besides the data layer (by creating UserResources, ShortResources, BlobResources and working around those) . At first glance, it looked like it could be a way to go with this project, but then we read the assignment again and realized that we could not change the app layer. (this methodology would confuse and probably make the implementation harder down the line)

Although the delivery date was postponed, the overload of evaluations during this period prevented us from devoting more time to perfecting the work. As a result, we achieved the functional level required to cover the functionalities rated up to 15, but were unable to move on to the additional functionality requirements.

Notes:

We leveraged AI tools to enhance the quality of our english/text of our report