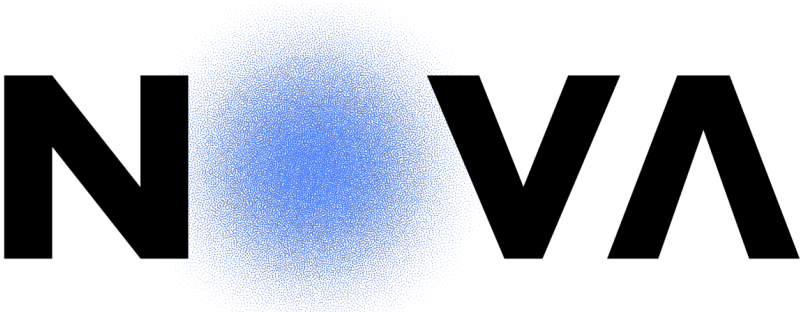

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Sistemas de Computação

em Cloud

Project 1

João Zarcos 60183

Miguel Domingos 60431

Azure PaaS

In developing the backend of the Tukano app, the group has leveraged a range of Azure resources to manage data storage, caching, and application hosting efficiently.

The core Azure resources utilized are:

- **Azure App Service Plan** - This service plan hosts and scales the application, allowing us to adjust resources as demand changes, ensuring the application performs optimally even under high user traffic.
- **Azure App Service** - This service hosts the web application itself, providing a fully managed platform that supports multiple programming languages and integrates easily with other Azure resources.
- **Azure Cosmos DB for PostgreSQL Cluster** - This service allows for fast, flexible querying and horizontal scalability, which is crucial for handling the massive amounts of user-generated content typical of a social network.
- **Azure Cosmos DB Account** - This account manages the Cosmos DB instances used for data storage and retrieval, enhancing data access speeds and availability.
- **Azure Cache for Redis** - An in-memory data store, Azure Redis Cache reduces load times by caching frequently accessed data, which is essential for quick response times and reducing strain on the primary database.
- **Azure Storage Account** - Primarily used to store user-generated content, like video files, Azure Storage is a reliable, scalable storage solution that supports large files and binary data and integrates with other Azure services for seamless data management.

The group has some things to point out about the work developed. Firstly, in terms of our caching strategy, we utilized the cache across the different types of data:

- **Users** - All user data is cached except for searches. User data is frequently accessed, so caching improves retrieval speed significantly. However, search functionality is not cached due to the large variety of possible search patterns. Since users rarely search for the same pattern repeatedly, caching would have minimal impact and could introduce unnecessary complexity.
- **Shorts** - Most data related to video content is cached, with the exception of the feed retrieval. The feed can change rapidly because users follow numerous creators who post content frequently. Caching the feed would lead to frequent cache updates, reducing the cache's efficiency. Therefore, it's more effective to retrieve the feed dynamically to ensure users see the latest content.
- **Blobs** - All media files are cached to optimize access speed and reduce latency. Since media files are large and accessed frequently, caching them reduces storage retrieval times and enhances the user experience.

Secondly, to ensure the testing of the code runs smoothly, it is needed for the professor to create a folder named “resources” and insert there the file “azurekeys-region.props”. Inside that file, the professor should copy and paste the keys:

```
BlobStoreConnection=

COSMOSDB_KEY=

COSMOSDB_URL=

COSMOSDB_DATABASE=

REDIS_HOSTNAME=

REDIS_KEY=

COSMOSDB_SQL_TYPE=

COSMOSDB_POSTGRES_URL=

COSMOSDB_POSTGRES_USER=

COSMOSDB_POSTGRES_PASSWORD=
```

The tester, then, inserts the values associated with the keys, keeping in mind that the **COSMOSDB_SQL_TYPE** can take 2 values:

- **COSMOSDB_SQL_TYPE=P** , if you want to use the CosmosDB for PostgreSQL
- **COSMOSDB_SQL_TYPE=N** , if you want to use the CosmosDB for NoSQL

In terms of implementation, the team decided to create two interfaces for processing both Users and Shorts, which would vary based on the database type used. Specifically, we implemented two classes for Users and two for Shorts, with one version tailored for a NoSQL database and the other for PostgreSQL. We also developed a dedicated class to handle the connection to the NoSQL database. For PostgreSQL, we used the pre-existing "DB" class provided by the instructors, which leverages Hibernate for ORM (Object-Relational Mapping) functionality; we updated this class to better suit our requirements.

Additionally, we standardized the naming convention for all identifier variables, changing terms like shortId and userId to simply id, and updated all corresponding getter and setter methods accordingly. This change simplifies our data-handling processes, making it easier and more efficient to store and retrieve data across different databases.

Performance Evaluation of Solution

Cache VS No cache

In this case, we are comparing the backend using a cache VS without using a cache, utilizing the CosmosDB PostgresSQL.

The performance tests show that the implementation of caching resulted in significant improvements in several operations. For user registration, the average response time improved by 42.9 ms, representing a 41.7% improvement. The P95 response time improved by 87.5 ms, or 36.9%. For the upload shorts operation, the average response time reduced by 74.5 ms, representing a 33.2% improvement, while the P95 response time improved by 157.8 ms (31.1%). In the realistic flow, the improvement was more modest, with the average response time improving by 17.8 ms (26.2%) and the P95 response time improving by 12.4 ms (14.2%). Finally, the user deletion operation had minimal improvement, with the average response time improving by only 4.7 ms (5%) and the P95 improving by 5.3 ms (4.4%).

In summary, user registration and upload shorts operations benefited the most from caching, with response time reductions of up to 42.9 ms and 157.8 ms, respectively. User deletion, on the other hand, saw minimal benefit, with improvements around 5%.

NoSQL VS PostgreSQL

Based on the performance data that the group gathered, PostgreSQL is demonstrating stronger performance and stability compared to CosmosDB NoSQL. Here are the reasons:

- Across both realistic flow and delete operations, PostgreSQL consistently achieves lower response times than NoSQL. For instance, in the realistic flow, PostgreSQL's average response time (2045 ms) is higher than NoSQL (374.7 ms), but PostgreSQL’s P95 and P99 metrics show it can handle the workload more predictably under peak loads (P95: This value indicates that 95% of the requests were processed faster than this latency, and only 5% had a higher latency; P99: 99% of the requests were processed faster, and 1% had a higher latency).

- PostgreSQL sessions typically finish faster than NoSQL sessions, meaning that each user request requires less time to process end-to-end. In the delete user flow, for instance, PostgreSQL’s session length has a median of 301.9 ms, whereas NoSQL is higher at 596 ms, reflecting more efficient use of resources in PostgreSQL.

- NoSQL handled the load without failures during the delete user test, which suggests strong reliability in certain low-frequency operations. However, PostgreSQL’s overall error rates across endpoints in the realistic flow are generally lower, indicating fewer disruptions in continuous usage.

Summary

This Azure-based architecture establishes a robust foundation for scaling the Tukano app. By leveraging Azure's managed services and strategically implementing caching, we have optimized performance, reduced costs, and ensured the platform can efficiently handle large volumes of user-generated content. This approach allows us to deliver a highly responsive, user-friendly experience that meets the demanding expectations of our social media users.

By choosing managed services, we also simplify infrastructure management, enabling the team to focus more on feature development and less on maintenance. The caching strategy, using Azure Cache for Redis, helps reduce latency and offloads frequent requests from the main database, further enhancing scalability and speed. Additionally, Azure’s autoscaling capabilities ensure the platform can adjust resources dynamically, handling traffic spikes during peak times without compromising user experience.

Overall, this architecture is designed for resilience, cost-effectiveness, and seamless scalability, providing Tukano with the agility to grow as its user base expands.