Sistemas de Computação em Cloud / Cloud Computing Systems

2024/2025

Project Assignment 1

Autores: Daniel Pojega, 71911

Tiago Neutel, 71903

10 novembro 2024

Índice

1. Introduction / Project Overview

This project is a web application inspired by popular video-sharing platforms like TikTok and YouTube Shorts. The primary goal is to improve our understanding of working with cloud services and managing cloud-based architectures.

The application is structured around a RESTful server, represented by the TukanoRestServer class, which handles all incoming requests to the application. This server then routes each request to designated classes based on functionality, including User, Short, Blobs, Likes, and Following. Each of these components is responsible for interacting with specific Azure services for data storage and management, using Azure Blob Storage for media files, Cosmos DB for database management, and Redis Cache for caching.

While we aimed to meet all proposed objectives, some were outside our current implementation scope. The application is currently operational only in a local environment, without SaaS deployment or geo-replication. We successfully implemented Cosmos DB with the NoSQL configuration and made progress on configuring Cosmos DB for PostgreSQL, although this remains untested. Despite the lack of verification, we are optimistic that the Cosmos DB PostgreSQL setup should be functional.

2. Design

2.1. Changes to the project

During the development of the *Tukano* project, we identified a few areas in the original Tukano API that required adjustments to better align with our project requirements and cloud architecture. These enhancements allowed us to comprehend and improve the API to the specific needs of our implementation, ensuring a smoother integration with Azure services and improving overall functionality.

An important adaptation was the separation of the RESTFollowing and RESTLikes interfaces from the original RESTShorts interface. Previously, these functionalities were part of the RESTShorts interface, but this approach was reconsidered as we believe that it didn't follow the natural structure of the data.

By moving RESTFollowing and RESTLikes into their own interfaces, we established a clearer separation of concerns, which better reflects the distinct roles these entities play within the application. This decision improves the overall organization of the code, even though it requires additional containers in the database to store and manage these entities independently from the Shorts entity. This trade-off was deemed worthwhile, as it aligns the application structure with real-world relationships among shorts, followers, and likes.

The addition of these two interfaces, meant that the following classes and interfaces had to be changed/added: Short, Shorts, User, FollowingI, LikesI, RestShortsResources, RestLikesResources, RestFollowingResources.

2.2. Database implementation

We used Azure Blob Storage as our primary solution for handling blobs, specifically for storing and managing large media files, such as images and videos, which are central to our project's functionality. Blob Storage is optimized for unstructured data, making it an ideal choice for media content that doesn't require a rigid schema. This approach allows us to leverage Azure's scalability and redundancy features, ensuring efficient storage and reliable access to media files.

The Azure Blob Storage implementation is encapsulated in the AzureBlobStorage class, replacing the original FileSystemStorage class and implementing the BlobStorage interface as specified in the original code. The main adaptation is in the write function, which no longer requires a specific path, as all data is stored within the same container (BlobsContainer). The remaining operations function similarly, but they now interact with Azure Blob Storage instead of the file system. This implementation largely builds on concepts and techniques explored in our laboratory classes.

We used Azure Redis Cache and Azure Cosmos DB as key components in our application's data management strategy, each serving a distinct role. Redis Cache was chosen for its ability to provide quick access to frequently used data, while Cosmos DB served as the main database for persistent data storage.

Redis Cache was implemented in the RedisCache class and acts as an in-memory cache, storing recently accessed items for rapid retrieval. For instance, in the JavaUsers class, we cache user data upon retrieval to avoid redundant database queries, allowing future requests for the same user to be served directly from Redis. This approach significantly reduces access times for popular items and minimizes load on the database. Additionally, caching query results, such as user search patterns, enhances performance in scenarios involving multiple users accessing similar data.

Cosmos DB serves as the primary database, implemented through the CosmosDB class. By default, each CRUD operation interacts with Cosmos DB to maintain the latest state of data. For example, when a user is created, updated, or deleted, the operation is handled directly through Cosmos DB, ensuring that all data modifications are saved reliably. In cases where cached data is updated, such as in user updates or deletions, Cosmos DB ensures that the underlying data remains consistent, while the cache is refreshed to reflect changes.

In our implementation, each class is configured to interact with a specific container within Cosmos DB, ensuring that data is organized and accessible by function. For example, the JavaUsers class connects to the "users" container to manage user-related data, while other classes, such as JavaShorts and JavaBlobs, connect to their respective containers. This separation allows each container to be optimized for its unique data requirements,

improving query performance and making data management more modular. By isolating data into different containers, we achieve better scalability and maintainability across various parts of the application.

As previously mentioned, this required a change to all the other classes that weren't the Blob, but only to redirect them to use Cosmos instead of Hibernate and to also make the verifications needed to use Redis Cache as well as to use it.

3. Performance evaluation

3.1. Tests

For our current code, individual tests were conducted for each endpoint. Initially, we used Postman to verify that each endpoint functioned as expected, with all tests passing successfully. Following these tests, we conducted performance testing with Artillery to evaluate each endpoint individually.

To avoid taking up excessive space and repetition of screenshots, only the response times for each test will be included.

Endpoint	Average Time (ms)	Mean Time (ms)
postUser	109	82
getUser	120	135.7
searchUsers	289	290
updateUser	76	76
deleteUser	77	76
createShort	247	347
deleteAllShorts	583	354
deleteShort	298	333
getFeed	278	202
getShort	311	327
getShorts	319	308
Like	494	496
getLikes	292	247
Follow	425	424
getFollowers	192	70

3.2. Provided tests

In this section, we present a performance evaluation based on the tests provided by the professor.

User_register.yaml:

```
http.codes.200: ..... 600
http.downloaded_bytes: ..... 47137
http.request_rate: ..... 6/sec
http.requests: ..... 600
http.response_time:
  min: ..... 59
  max: ..... 546
  mean: ..... 234.9
  median: ..... 267.8
  p95: ..... 376.2
  p99: ..... 441.5
```

Upload_shorts.yaml:

```
http.codes.200: ..... 100
http.codes.403: ..... 100
http.downloaded_bytes: ..... 23340
http.request_rate: ..... 5/sec
http.requests: ..... 200
http.response_time:
  min: ..... 2
  max: ..... 386
  mean: ..... 176.3
  median: ..... 21.1
  p95: ..... 361.5
  p99: ..... 376.2
```

Realistic_flow.yaml:

(This test was failing sometimes due to some JSON problem for unknown reasons, below goes one of the successful attempts made)

```
http.codes.200: ..... 6
http.codes.404: ..... 5
http.codes.500: ..... 10
http.downloaded_bytes: ..... 555
http.request_rate: ..... 5/sec
http.requests: ..... 21
http.response_time:
  min: ..... 2
  max: ..... 1293
  mean: ..... 622.2
  median: ..... 518.1
  p95: ..... 1249.1
  p99: ..... 1249.1
```

User_delete.yaml:

```
http.codes.204: ..... 200
http.downloaded_bytes: ..... 0
http.request_rate: ..... 2/sec
http.requests: ..... 200
http.response_time:
  min: ..... 249
  max: ..... 445
  mean: ..... 268.5
  median: ..... 262.5
  p95: ..... 284.3
  p99: ..... 347.3
```