NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Project Assignment #1

SCC 2024/25

David Castro 60973

Yaroslav Hayduk 60739

10th November

Table of Contents

Porting TuKano

Azure Blob Storage

Integrating Azure Blob Storage for TuKano was straightforward. The base application had a *FilesystemStorage* class where all the necessary operations for the blobs were implemented. To adapt to Azure, we created an *AzureStorage* class with similar functionality but implemented using Azure Blob Storage. In this class we initialize *BlobContainerClientBuild* using our Azure storage connection string, allowing us to perform standard operations like reading, writing and deleting blobs. It's also in this class where we implement our *triggerRead* function, which is used for the counting views feature, whose details will be explained later in this report.

Azure Cosmos DB

The integration with Azure Cosmos DB followed a similar approach to Blob Storage, where we have an interface with a list of database operations (the **DB** class) used by our application, and we wanted to keep these operations unchanged to keep consistency. We created the **CosmosDB** class with the implementations of the functions that leverage Cosmos DB for NoSQL, which will be used by the **DB** class. This solution allows us to keep the same logic as before throughout the application while only needing to slightly change the **DB** class. Within the **CosmosDB** class, we initialized our Cosmos database using Azure-specific information, such as database names and keys, to access containers by name. Additionally, we implemented a function to select the appropriate container dynamically for each operation by accepting a string parameter and switching containers based on it. For example, when a *persistOne* operation for a **User** object is invoked, this function ensures that the current container is set to "users" before execution. We implemented a similar method to switch containers based on query requirements, ensuring that the correct container is chosen for each query.

As for the CosmosDB for PostgreSQL, we took advantage of the already existing Hibernate class. Changes were limited to updating the hibernate.cfg.xml file to match the JDBC settings (such as driver, URL, username, and password) provided by our new resource. We also had to make minor adjustments in the structure of the queries to account for syntax differences. Overall, the biggest challenges we faced were due to queries syntax and container structure. At first, we didn't know how to efficiently change to the correct container based on our needs, then we got confused if we should have the classes "**likes**" and "**following**" on separate containers, or in the "**shorts**" container, given that CosmosDB for NoSQL queries do not support the union over different containers.

Azure Cache for Redis

To implement Azure's cache functionalities, we created a *RedisCache* class, just like it was done in the labs. We didn't take the time to explore and try out different parameters for initializing the cache pool as we felt our time was better spent developing other parts of the project. We did experiment with different values for the TTL of the different items in cache, although we weren't able to reach any meaningful conclusion, since for all the different combinations of values no significant changes were achieved.

We designed the cache to store users, shorts and their likes as we felt those were the main data domains that were going to be accessed in our database and therefore would be the most useful to have their access time reduced. Initially the likes were stored in cache as an integer that we increment and decrement based on the value of *isLiked*, however, upon further thought and analysis, we realized that this could lead to problems if requests were made where *isLiked* is set to the value it already has in the database, so the database would remain unchanged and the value in cache would be changed, incorrectly. To solve this, with the setup we had, we could try to retrieve the like from the database to verify if the value should indeed be incremented or decremented, but the cost of the operation would defeat the purpose of having a cache in the first place. Only now, upon better analyzing the code, did we realize we could have made it so the *deleteOne* method returns not just always an *ok*, but also an error in case the object wasn't in the database and use this information for the cache check.

Azure Functions

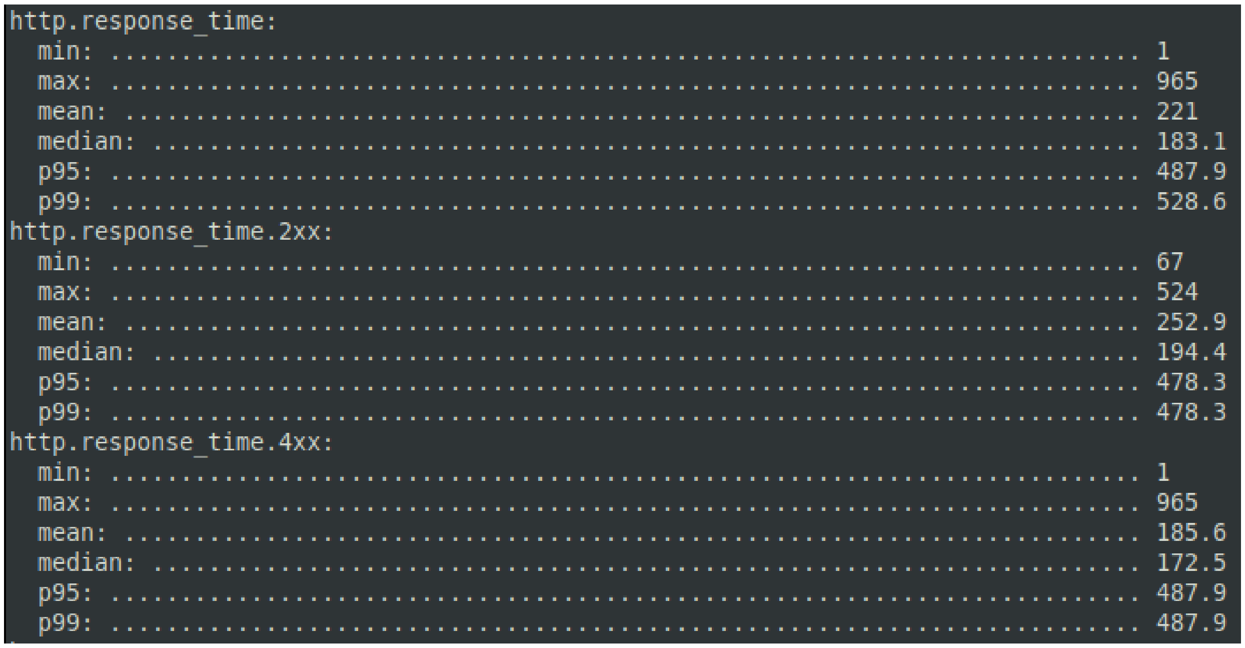
To implement the view-counting feature, we used Azure Functions, a serverless solution. Rather than counting views directly, this function increments the *totalViews* field of a "**short**" object each time the short is downloaded from Blob Storage. This approach is efficient because it avoids modifying multiple classes, such as **JavaBlobs** and **JavaShorts**, and removes the need for cross-class communication.
Our solution relies on an *HttpTrigger* that executes the view counting function whenever there is an HTTP request to download a blob. Query parameters are extracted from the URI to reconstruct the *shortId*, allowing the function to retrieve the corresponding **short** object from CosmosDB and update the *totalViews* field.
To ensure the function triggers on each blob download, we created a function that initiates with every read operation in Blob Storage (which is only used for downloads in this case). This function constructs the request URI by combining the Azure Functions' generated URI with the blob name as a query parameter and then sends an HTTP GET request to this URI on a separate thread.
This integration posed some challenges, particularly due to difficulties encountered during Lab 5 in practical lessons. The issues were primarily due to Java version mismatches and an incorrect execution order of provided resources. Additionally, we initially misunderstood the directory structure, attempting to place all management and function code within the TuKano directory, which led to deployment errors.

Performance Analysis

For the performance analysis we analyzed 4 scenarios using the tests provided by the professors. For the sake of comparison, we will be using the results of the realistic flow test.

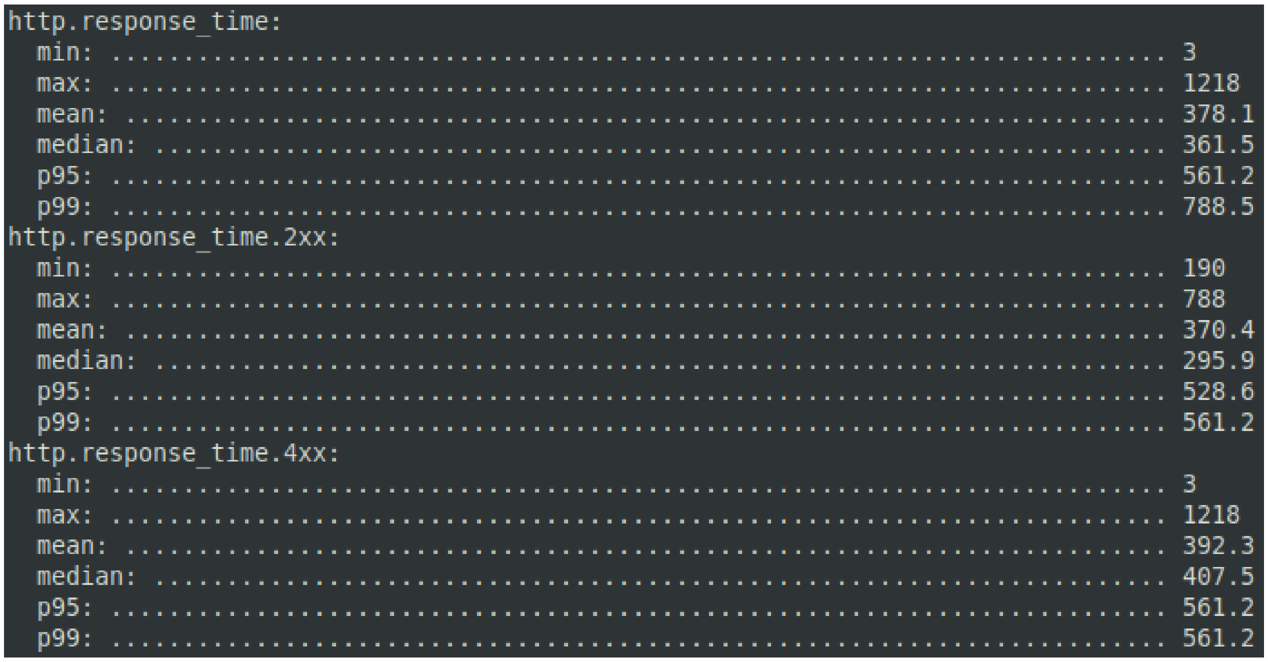
Azure Cosmos DB for NoSQL

Results of NoSQL with cache on:



```
http.response_time:
  min: ......................................................................... 1
  max: ......................................................................... 965
  mean: ........................................................................ 221
  median: ...................................................................... 183.1
  p95: ......................................................................... 487.9
  p99: ......................................................................... 528.6
http.response_time.2xx:
  min: ......................................................................... 67
  max: ......................................................................... 524
  mean: ........................................................................ 252.9
  median: ...................................................................... 194.4
  p95: ......................................................................... 478.3
  p99: ......................................................................... 478.3
http.response_time.4xx:
  min: ......................................................................... 1
  max: ......................................................................... 965
  mean: ........................................................................ 185.6
  median: ...................................................................... 172.5
  p95: ......................................................................... 487.9
  p99: ......................................................................... 487.9
```

Results of NoSQL with cache off:



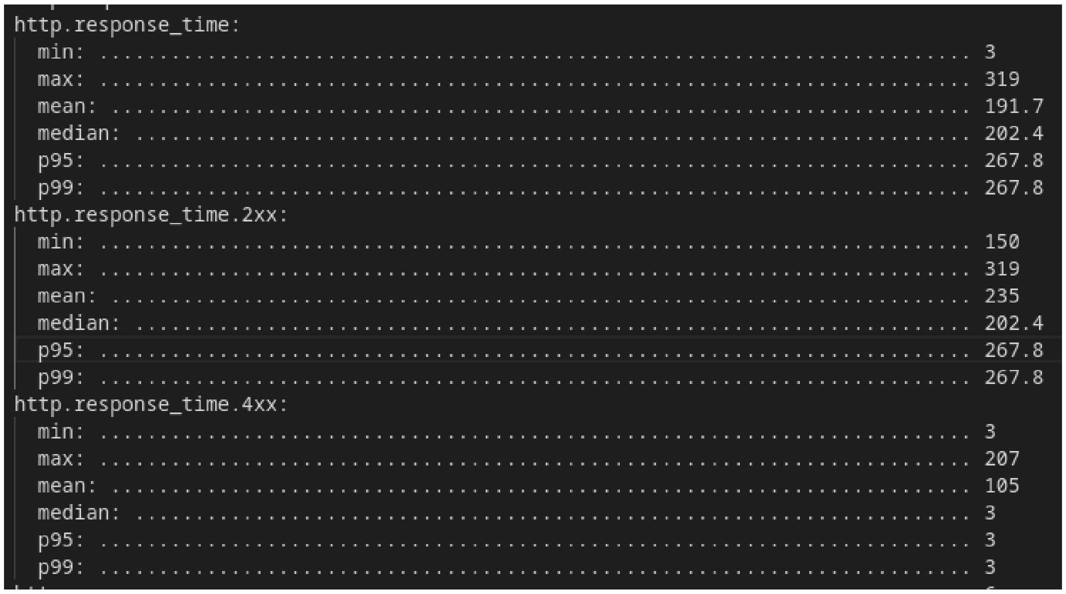
```
http.response_time:
  min: ......................................................................... 3
  max: ......................................................................... 1218
  mean: ........................................................................ 378.1
  median: ...................................................................... 361.5
  p95: ......................................................................... 561.2
  p99: ......................................................................... 788.5
http.response_time.2xx:
  min: ......................................................................... 190
  max: ......................................................................... 788
  mean: ........................................................................ 370.4
  median: ...................................................................... 295.9
  p95: ......................................................................... 528.6
  p99: ......................................................................... 561.2
http.response_time.4xx:
  min: ......................................................................... 3
  max: ......................................................................... 1218
  mean: ........................................................................ 392.3
  median: ...................................................................... 407.5
  p95: ......................................................................... 561.2
  p99: ......................................................................... 561.2
```

Conclusion based on scenario evaluation

Leveraging the cache allowed Tukano to significantly decrease overall response time of the request, which consequently makes the app faster. We also tested with multiple different values of cache TTL for each entity, however there were no cases we could determine there was a significant improvement.

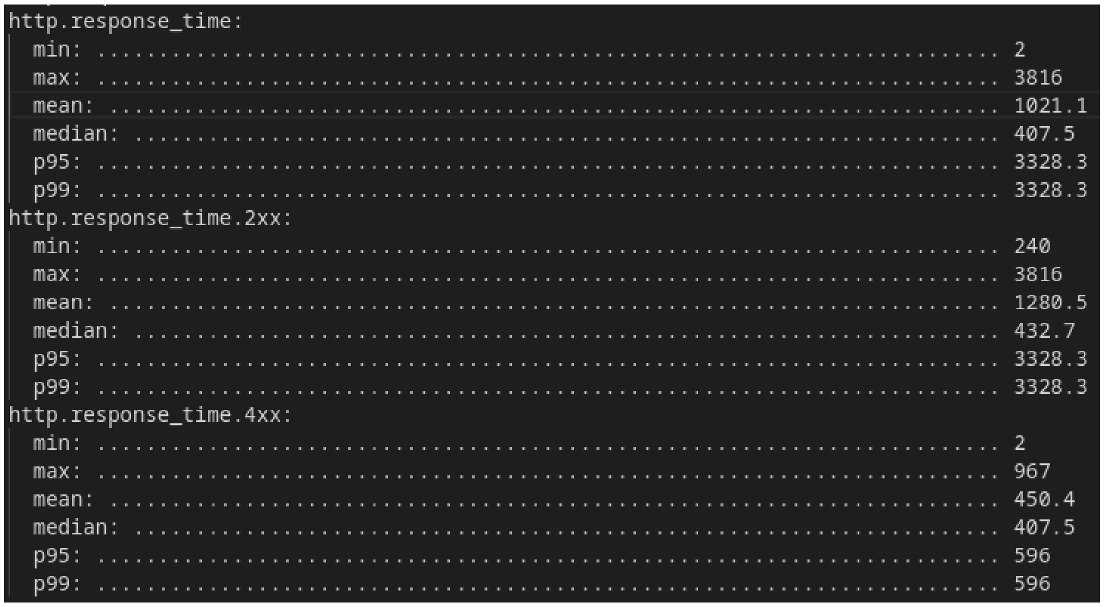
Azure Cosmos DB for PostgreSQL

Results of PostgreSQL with cache on:



```
http.response_time:
  min: ...................................................................... 3
  max: ...................................................................... 319
  mean: ..................................................................... 191.7
  median: ................................................................... 202.4
  p95: ...................................................................... 267.8
  p99: ...................................................................... 267.8
http.response_time.2xx:
  min: ...................................................................... 150
  max: ...................................................................... 319
  mean: ..................................................................... 235
  median: ................................................................... 202.4
  p95: ...................................................................... 267.8
  p99: ...................................................................... 267.8
http.response_time.4xx:
  min: ...................................................................... 3
  max: ...................................................................... 207
  mean: ..................................................................... 105
  median: ................................................................... 3
  p95: ...................................................................... 3
  p99: ...................................................................... 3
```

Results of PostgreSQL with cache off:



```
http.response_time:
  min: ...................................................................... 2
  max: ...................................................................... 3816
  mean: ..................................................................... 1021.1
  median: ................................................................... 407.5
  p95: ...................................................................... 3328.3
  p99: ...................................................................... 3328.3
http.response_time.2xx:
  min: ...................................................................... 240
  max: ...................................................................... 3816
  mean: ..................................................................... 1280.5
  median: ................................................................... 432.7
  p95: ...................................................................... 3328.3
  p99: ...................................................................... 3328.3
http.response_time.4xx:
  min: ...................................................................... 2
  max: ...................................................................... 967
  mean: ..................................................................... 450.4
  median: ................................................................... 407.5
  p95: ...................................................................... 596
  p99: ...................................................................... 596
```

Conclusion based on scenario evaluation

As expected, the use of SQL queries significantly increased the response time of the app. That behavior is expected due to the way both databases scale.
NoSQL databases are designed to scale horizontally, meaning they can distribute data across multiple servers and add more nodes as demand grows (sharding).

While SQL databases are usually scaled vertically, meaning to improve performance we will have to add more resources to a single server. Given that we chose the cheapest option for our PostgreSQL database (Single node, Burstable, 1 vCore, 2 GiB RAM and 32 GiB Node Storage) the expected performance wasn't that great.

NoSQL vs SQL

When we analyze our results for a normal flow of the app, we can see that the overall best results were obtained by using PostgreSQL with cache on. Even though we get very similar mean and median results with NoSQL, when we look at the p95 and p99 metrics, PostgreSQL vastly outperforms NoSQL. This means that, for NoSQL, there is a bigger disparity in response times, either requests are very fast or very slow, while for PostgreSQL they are more uniform, giving very similar results for 99% of requests. This can also be seen by looking at the max response time of a request, where NoSQL's is three times bigger. This difference is most likely explained by operations like *deleteAllShorts* because, since NoSQL does not support delete queries, we are required to make a select query for all the values we want to delete and then delete them one by one, whereas in PostgreSQL we can delete everything we want in a single query. The *getFeed* operation is also harder to do in NoSQL. We made the decision to store every entity in their own container, that is, shorts go to the *shorts* container, followings go to the *following* container and so on. Due to this, to get a user's feed, we need to first query all the users he's following, build the string from the resulting list to use in the '*IN*' clause, and then make a second query to get all their shorts. Considering all these observations, these were the expected results.

Observations

Our mistakes

The day after the delivery, while completing our report and running the artillery tests, there were multiple tweaks that we noticed could've been made to our code to increase its quality, but there was one major mistake in particular that we would like to bring attention to, which is in class *JavaShorts*, in the method *deleteAllNoSqlShorts*. The return value, instead of 'DB.transaction', should be 'DB.noSqlTransaction'. We aren't sure if the professors are going to try to run the code, but we would like to make that clarification in case you do, since such a silly oversight, probably during the merging of branches, causes the *deleteAllShorts* function to stop working for NoSQL.

On the report delivery day, while trying to obtain the performance test results for PostgreSQL database, for some unknown reason the resource wasn't working properly, we ended up not being able to obtain the performance results for that scenario. Therefore, we opted to submit the report late. The issue is still unknown, considering that we deleted the resource and recreated it (in the same manner as before) and only then were we able to send requests to that database.

AI Uses

Our group did get assisted by AI, but not in the sense of directly asking for code and doing most of the work for us. It was more about getting information on certain topics, debugging and error fixing.
Examples of uses are:

- Figuring out how to switch containers in the cosmosDB.
- Some issues we had in the *deleteUser* method. AI recommended creating a method that fetches all blobs within a directory and deletes them iteratively, the previous solution tried to delete the folder for the user which was not possible.
- Fixing the PostgreSQL database use, mostly explaining the issues within the hibernate.cfg.xml file and recommending solutions.
- Explained the appropriate directory structure for the Azure Functions deployment.
- Fixing some SQL queries.

The most direct and only uses of code generated by AI are related to the Azure Functions, in our case we had problems figuring out how to make a read over the Azure Storage trigger a serverless function, so subsequently some of the code in the *triggerRead* method in the AzureStorage was “inspired” by AI.