NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Cloud Computing Systems

2024/2025

1st Project Report - Azure Tukano

Authors: **Lab class Nº** P3

70526, André Branco
70527, João Silveira

Assignment Analysis

First impressions

Our first impressions of the project were somewhat clear from the start. Having previously worked with the ‘Tukano’ application in the subject of ‘Distributed Systems’, we knew what we were in for.

Source code

The given source code was using the File System, on the machine that it ran on, for blob storage and the Hibernate sql tables (being its default when not set to an indicated storage system). And also noticed there was no cache system implemented in the app, from here we recognized our main changes and additions were going to be adding a new cache system, blob storage modifications, Cosmos DataBase related classes, and alterations to the JavaUsers, JavaShorts and JavaBlobs to use these new storage systems.

Addition of Cloud Storage Systems to the source code

For most of our new additions and modifications to the code, many different dependencies had to be added in the **pom.xml** file, most of which came from the Lab classes.

- Cache System Classes

  Redis Cache - This was mostly taken from Lab4, where we learned to work with Azure Cache for Redis, and it creates/returns the client pool for cache access.

  CacheForCosmos - This is the layer that makes use of the client pool, and makes the necessary changes to the cache, be it an insert, update, delete or get call. Also having time-to-live (TTL), on objects inserted, and refreshing the time upon get calls, defined by a boolean to also control which items should or shouldn't be refreshed.

- ## NoSQL Classes

  CosmosDB_NoSQL - Also mostly taken from what was learned in Lab3, having its major differences on the choosing of the container for all methods.

  CosmosDB - This class is a layer for CosmosDB_NoSQL, like the ‘DB’ is for Hibernate, where, in this case, all interactions with CosmosDB_NoSQL happen here, avoiding making direct calls to it’s instance everywhere where an insert, update, delete or get would happen, and better handling of the result output.

- ## PostgreSQL

  For this storage system no new classes were needed, only changes to existing ones.

  hibernate.cfg.xml - Changes made to the session factory connection, having to change/add driver, url, username and password for it to connect to our CosmosDB PostgreSQL. Also added a dialect configuration, not mandatory, but useful for our debugging.

- ## Blob Service - Storage Account

  AzureBlobStorage - Similar to what was done in the practical class, but with extra restrictions added to ensure correct behavior of the application with the cloud storage, such as checking the existence of a blob before creating a new one, and if it exists checking if the contents are the same.

‘Java’ changes (Users, Shorts, Blobs)

Classes JavaUsers, JavaShorts and JavaBlobs had many changes due to the application now having a new storage system. A lot of references had to be made, and in some cases, new logic to the methods. Most changes in logic are around the same, so we’ll cover it as a general change, and not all specific methods, to not repeat too much.

- ### JavaUsers and JavaShorts

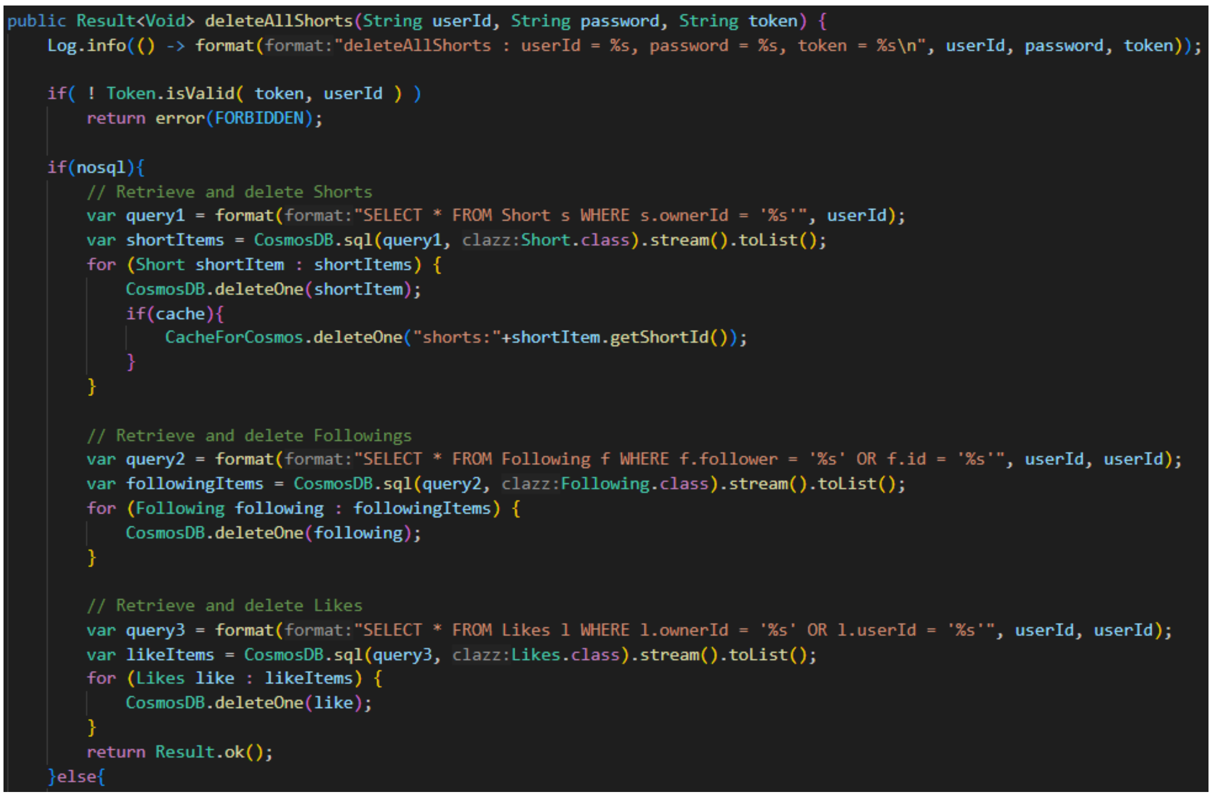
For easier change of systems and testing, we adapted most of the methods to depend on 2 environmental variables we created, ‘cache’ and ‘NoSQL’. These control whether the application uses cache or not, and the same goes for using NoSQL or PostgreSQL. So all public methods used by a client, either to insert, update, delete, or get, had to get this change. In most cases, we added ‘if’ operations to control these outcomes, so an ‘if’ for storing/using cache or not, and another ‘if’ for using either NoSQL or PostgreSQL.

These changes weren’t direct since there had to be some carefulness involved with SQL queries and the cache usage. To start, the queries for NoSQL and PostgreSQL are different since Postgre is a relational database and NoSQL isn’t. This means that Postgre can take more complex SQL queries, and also it has the capability of keeping its calls atomic, consistent, isolated, and durable (ACID), also allowing for ‘transactions’ where multiple requests can be executed into one, where essentially it can process requests for two or more tables at the same time. For NoSQL, being a non-relational database, most complex SQL queries had to be broken into separate calls, breaking the atomicity of the methods and making these slower.

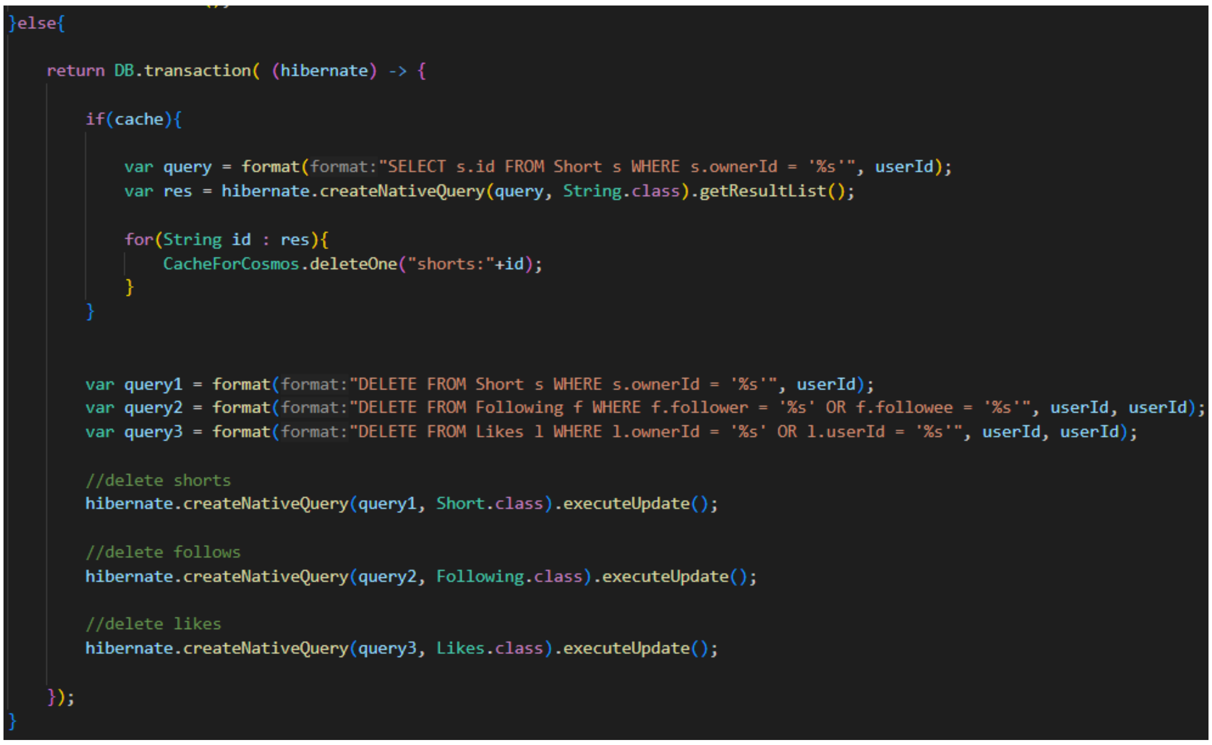
Overall, the changes ended up being that the calls to all three systems (cache, NoSQL, and PostgreSQL+Hibernate) were dependent on the boolean variables. Most SQL queries were changed since they’d be different depending on the database and the handle of cache misses, calling the database and inserting it back.

Here are some examples of those changes:

DeleteAllShorts(userId, password, token)

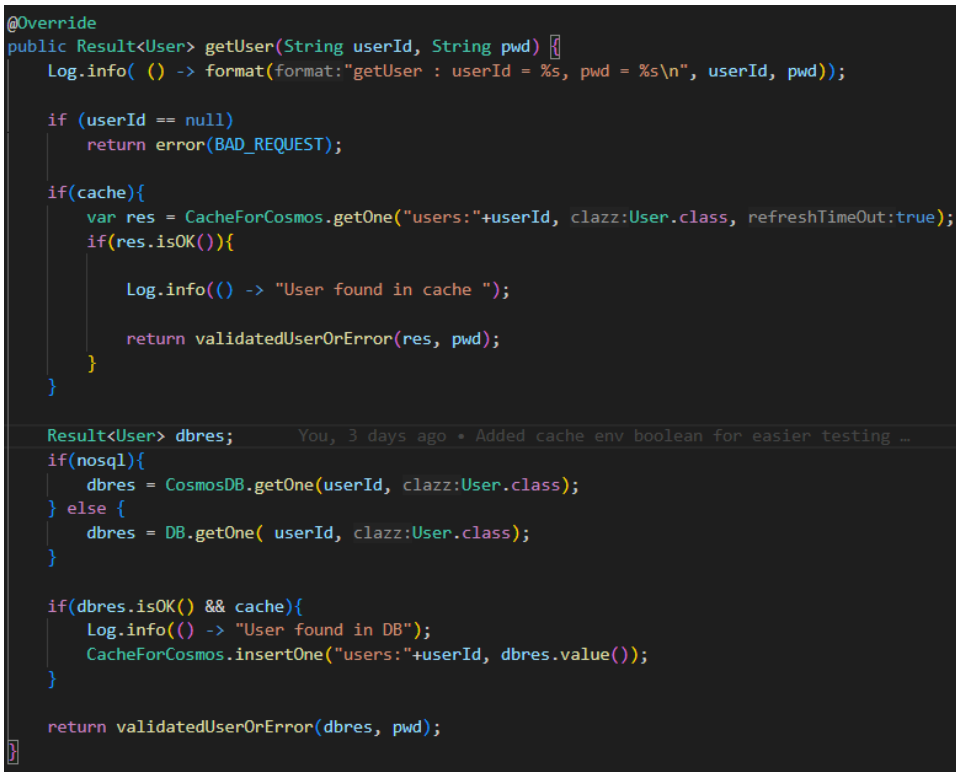


```java
public Result<Void> deleteAllShorts(String userId, String password, String token) {
    Log.info(() -> format(format:"deleteAllShorts : userId = %s, password = %s, token = %s\n", userId, password, token));

    if( ! Token.isValid( token, userId ) )
        return error(FORBIDDEN);

    if(nosql){
        // Retrieve and delete Shorts
        var query1 = format(format:"SELECT * FROM Short s WHERE s.ownerId = '%s'", userId);
        var shortItems = CosmosDB.sql(query1, clazz:Short.class).stream().toList();
        for (Short shortItem : shortItems) {
            CosmosDB.deleteOne(shortItem);
            if(cache){
                CacheForCosmos.deleteOne("shorts:"+shortItem.getShortId());
            }
        }

        // Retrieve and delete Followings
        var query2 = format(format:"SELECT * FROM Following f WHERE f.follower = '%s' OR f.id = '%s'", userId, userId);
        var followingItems = CosmosDB.sql(query2, clazz:Following.class).stream().toList();
        for (Following following : followingItems) {
            CosmosDB.deleteOne(following);
        }

        // Retrieve and delete Likes
        var query3 = format(format:"SELECT * FROM Likes l WHERE l.ownerId = '%s' OR l.userId = '%s'", userId, userId);
        var likeItems = CosmosDB.sql(query3, clazz:Likes.class).stream().toList();
        for (Likes like : likeItems) {
            CosmosDB.deleteOne(like);
        }
        return Result.ok();
    }else{
```



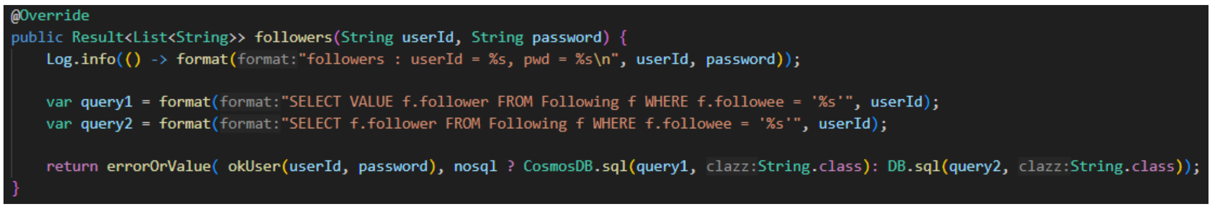
```java
}else{

    return DB.transaction( (hibernate) -> {

        if(cache){

            var query = format(format:"SELECT s.id FROM Short s WHERE s.ownerId = '%s'", userId);
            var res = hibernate.createNativeQuery(query, String.class).getResultList();

            for(String id : res){
                CacheForCosmos.deleteOne("shorts:"+id);
            }
        }


        var query1 = format(format:"DELETE FROM Short s WHERE s.ownerId = '%s'", userId);
        var query2 = format(format:"DELETE FROM Following f WHERE f.follower = '%s' OR f.followee = '%s'", userId, userId);
        var query3 = format(format:"DELETE FROM Likes l WHERE l.ownerId = '%s' OR l.userId = '%s'", userId, userId);

        //delete shorts
        hibernate.createNativeQuery(query1, Short.class).executeUpdate();

        //delete follows
        hibernate.createNativeQuery(query2, Following.class).executeUpdate();

        //delete likes
        hibernate.createNativeQuery(query3, Likes.class).executeUpdate();

    });
}
```

GetUser(userId, pwd)



```java
@Override
public Result<User> getUser(String userId, String pwd) {
    Log.info( () -> format(format:"getUser : userId = %s, pwd = %s\n", userId, pwd));

    if (userId == null)
        return error(BAD_REQUEST);

    if(cache){
        var res = CacheForCosmos.getOne("users:"+userId, clazz:User.class, refreshTimeOut:true);
        if(res.isOK()){

            Log.info(() -> "User found in cache ");

            return validatedUserOrError(res, pwd);
        }
    }

    Result<User> dbres;       You, 3 days ago • Added cache env boolean for easier testing …
    if(nosql){
        dbres = CosmosDB.getOne(userId, clazz:User.class);
    } else {
        dbres = DB.getOne( userId, clazz:User.class);
    }

    if(dbres.isOK() && cache){
        Log.info(() -> "User found in DB");
        CacheForCosmos.insertOne("users:"+userId, dbres.value());
    }

    return validatedUserOrError(dbres, pwd);
}
```

Followers(userId, password)



```java
@Override
public Result<List<String>> followers(String userId, String password) {
    Log.info(() -> format(format:"followers : userId = %s, pwd = %s\n", userId, password));

    var query1 = format(format:"SELECT VALUE f.follower FROM Following f WHERE f.followee = '%s'", userId);
    var query2 = format(format:"SELECT f.follower FROM Following f WHERE f.followee = '%s'", userId);

    return errorOrValue( okUser(userId, password), nosql ? CosmosDB.sql(query1, clazz:String.class): DB.sql(query2, clazz:String.class));
}
```

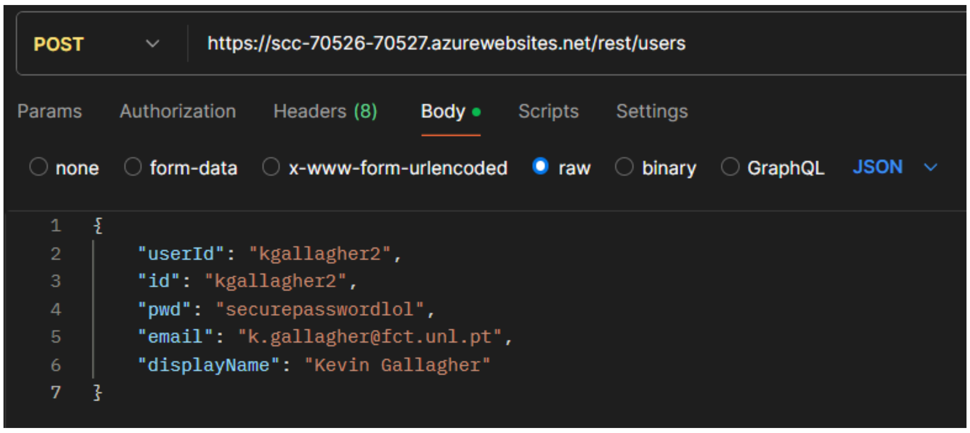
Alterations to the source code

- ### Added parameter ‘id’ to Object classes

This alteration was made because Azure’s Cosmos DB containers require an ‘/id’ parameter (with that specified name). We found different approaches to this problem, using “@JsonProperty(“id”) on the object’s already existing id, for example, the ‘userId’, so it would represent that ‘id’ parameter when referring to the object as a Json, or adding another “get” method on the class, with just “getId”, returning the object’s id. Both these approaches seemed to work for the insert of the objects in the cache and CosmosDB, but they started having many problems with the ‘getOne’ operations, decoding and converting from JSON to the object class and it wasn’t working well.

So to not lose time early in development, since we had to do a lot of other implementations, we ‘temporarily’ added a new parameter ‘id’ to the object classes in use, and were planning on changing it later when the rest of the project was mostly done and working. This ended up not happening, as we preferred guaranteeing that the application worked on both approaches of Storage systems and Cache, then changing the object classes to not have an ‘extra’ id.

Therefore when executing our method ‘createUser’ it is necessary to also add a new ‘id’ parameter to the JSON body like so:


POST https://scc-70526-70527.azurewebsites.net/rest/users

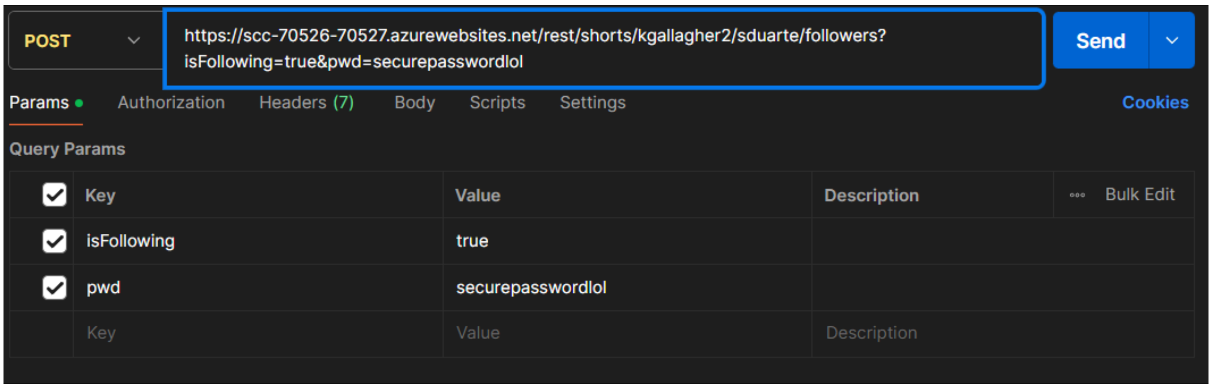
```
{
    "userId": "kgallagher2",
    "id": "kgallagher2",
    "pwd": "securepasswordlol",
    "email": "k.gallagher@fct.unl.pt",
    "displayName": "Kevin Gallagher"
}
```

Image 1: Post User example

The parameters ‘userId’ and ‘id’ **MUST** be the same for the application to work correctly.

- Boolean values for Like and Follow

  The boolean values to signify if a Like or Follow should be created or deleted (isLiked and isFollowing respectively) are now passed as a query parameter instead of in the body of the request. This was done for ease of use in testing and better readability.

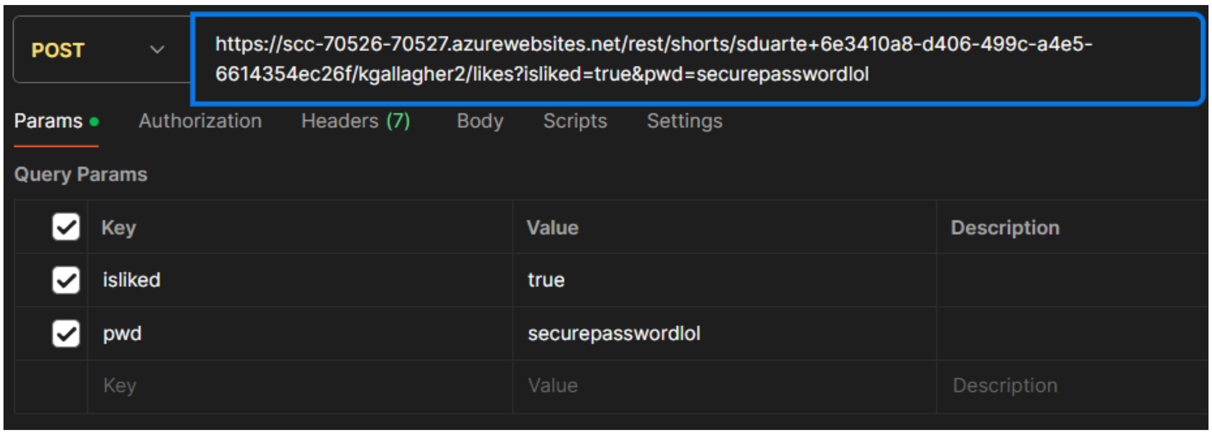


```
https://scc-70526-70527.azurewebsites.net/rest/shorts/kgallagher2/sduarte/followers?
isFollowing=true&pwd=securepasswordlol
```

Query Params

	Key	Value	Description
✓	isFollowing	true	
✓	pwd	securepasswordlol	

Image 2: Follow example



```
https://scc-70526-70527.azurewebsites.net/rest/shorts/sduarte+6e3410a8-d406-499c-a4e5-
6614354ec26f/kgallagher2/likes?isliked=true&pwd=securepasswordlol
```

Query Params

	Key	Value	Description
✓	isliked	true	
✓	pwd	securepasswordlol	

Image 3: Like example

Tests and Conclusions

For this section we used our own artillery tests, having them focused on comparing performance and latency between the cloud storage systems used and cache. We tested both NoSQL and PostgreSQL, with and without cache.

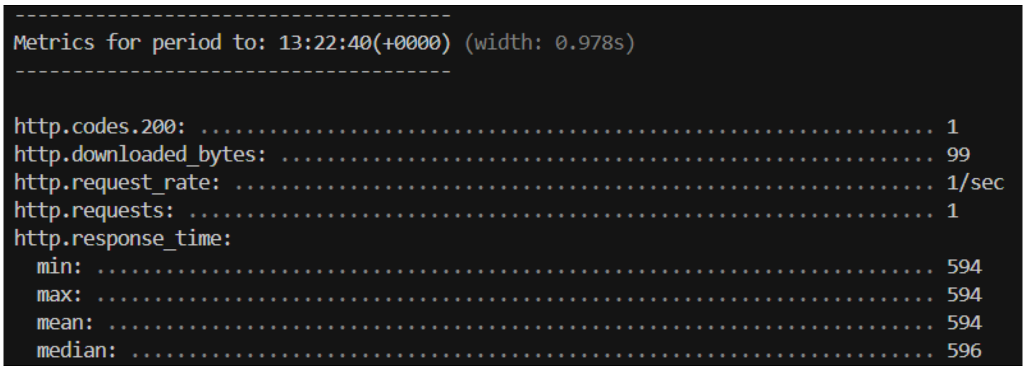
Most **functionality** tests were made in 'Postman', covering all methods and having constant debugging, throughout the code development.

For the **performance and latency** tests, not all methods had to be covered, as conclusions and numbers would repeat themselves on similar methods. Therefore we only tested more relevant methods, still covering inserts, updates, deletes and gets.

Cache vs no Cache

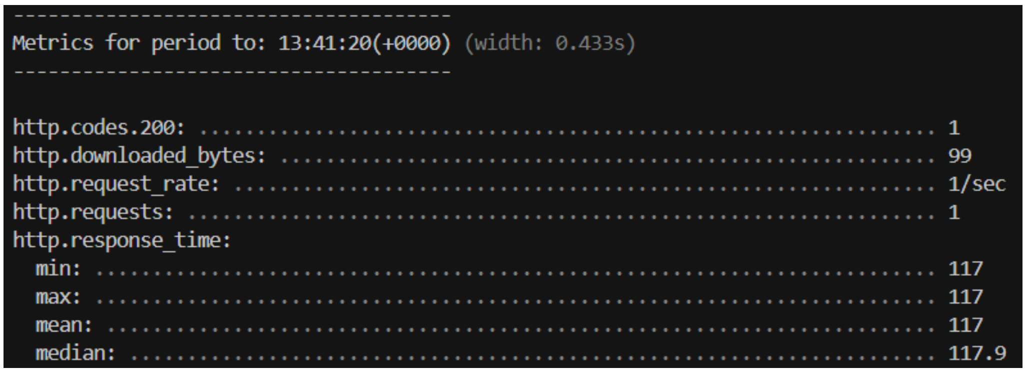
The cache has a TTL of 30s, which has the possibility of being refreshed when a cache hit occurs.
The main benefits with using cache or not, are in methods that require GET requests from the database, for example:

- getUser without cache



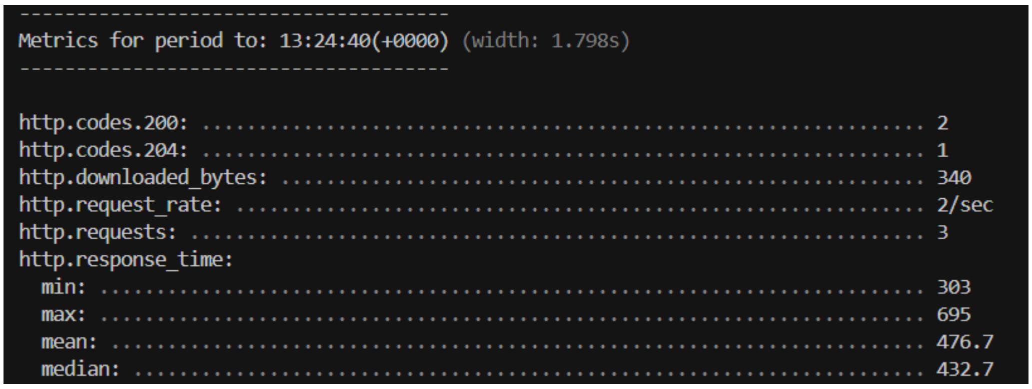
```
--------------------------------------
Metrics for period to: 13:22:40(+0000) (width: 0.978s)
--------------------------------------

http.codes.200: .............................................................. 1
http.downloaded_bytes: ....................................................... 99
http.request_rate: ........................................................... 1/sec
http.requests: ............................................................... 1
http.response_time:
  min: ....................................................................... 594
  max: ....................................................................... 594
  mean: ...................................................................... 594
  median: .................................................................... 596
```

- getUser with cache



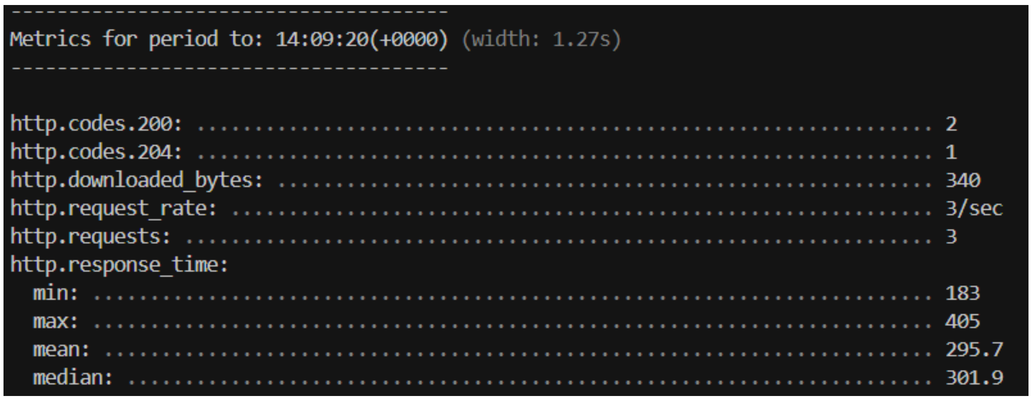
```
--------------------------------------
Metrics for period to: 13:41:20(+0000) (width: 0.433s)
--------------------------------------

http.codes.200: .............................................................. 1
http.downloaded_bytes: ....................................................... 99
http.request_rate: ........................................................... 1/sec
http.requests: ............................................................... 1
http.response_time:
  min: ....................................................................... 117
  max: ....................................................................... 117
  mean: ...................................................................... 117
  median: .................................................................... 117.9
```

- create short, like and get likes, without cache



```
-------------------------------------
Metrics for period to: 13:24:40(+0000) (width: 1.798s)
-------------------------------------

http.codes.200: ............................................................. 2
http.codes.204: ............................................................. 1
http.downloaded_bytes: ...................................................... 340
http.request_rate: .......................................................... 2/sec
http.requests: .............................................................. 3
http.response_time:
  min: ...................................................................... 303
  max: ...................................................................... 695
  mean: ..................................................................... 476.7
  median: ................................................................... 432.7
```

- create short, like and get likes, with cache



```
-------------------------------------
Metrics for period to: 14:09:20(+0000) (width: 1.27s)
-------------------------------------

http.codes.200: ............................................................. 2
http.codes.204: ............................................................. 1
http.downloaded_bytes: ...................................................... 340
http.request_rate: .......................................................... 3/sec
http.requests: .............................................................. 3
http.response_time:
  min: ...................................................................... 183
  max: ...................................................................... 405
  mean: ..................................................................... 295.7
  median: ................................................................... 301.9
```

Using these as our best examples, and while there were more tests run for cache, these give the same information we took from most tests. Focusing on the more meaningful parameters here, in the first example we have ‘https response time’ **mean** at 594ms (without cache) and 117ms (with cache), showing how much faster a cache hit is than a full request to the database.

For the second example, right after creating a Short, this one is inserted in the cache and the method ‘like’ makes a ‘getShort’ request, to verify this one’s existence. In the case of using cache, the cache hit shows how much faster the test can be, with **means** of 476.7ms (without cache) and 295.7ms (with cache), proving again that using cache is faster.

This upgrade to the app does have somewhat of a ‘weaker’ side, where when there’s insert requests it does take a little longer since it has to also insert the value in cache, but that insert time is so small that it’s about insignificant compared to the benefits on the ‘get’ requests.

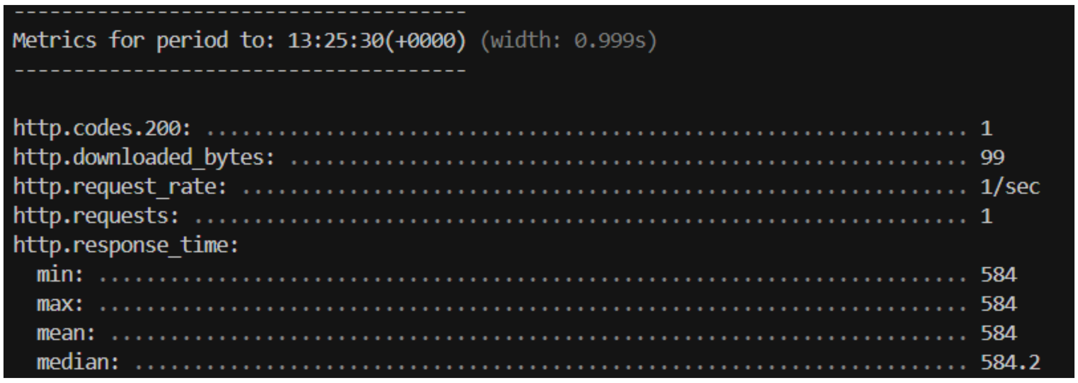
When performing many write operations on the database, such as creating users, the cache tends to slow down the application resulting in some timeouts.

NoSQL vs PostgreSQL

PostgreSQL is a relational database, while NoSQL isn't, and that gives it some advantages when comparing the two. (previously explained here)

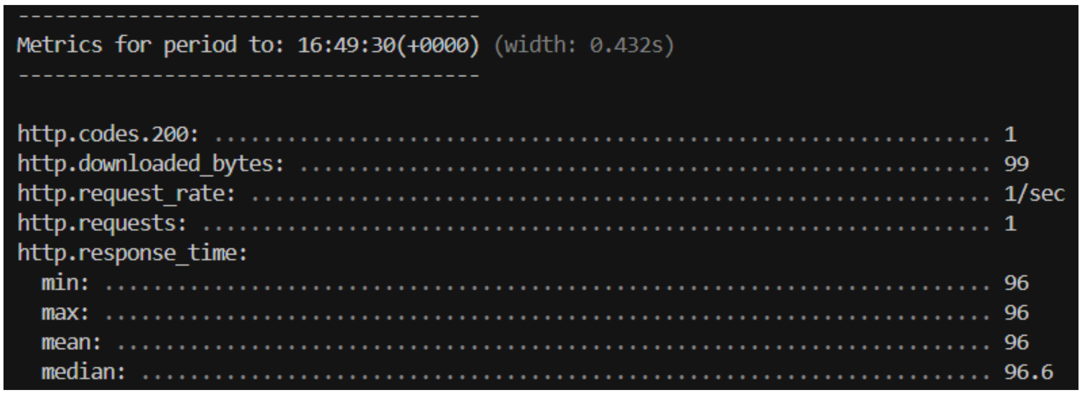
To reach a final conclusion on which is best between the two, and beyond just descriptions, we also took a more practical matter and tested both in relevant scenarios and compared their times.

- NoSQL without cache - Delete 1 user



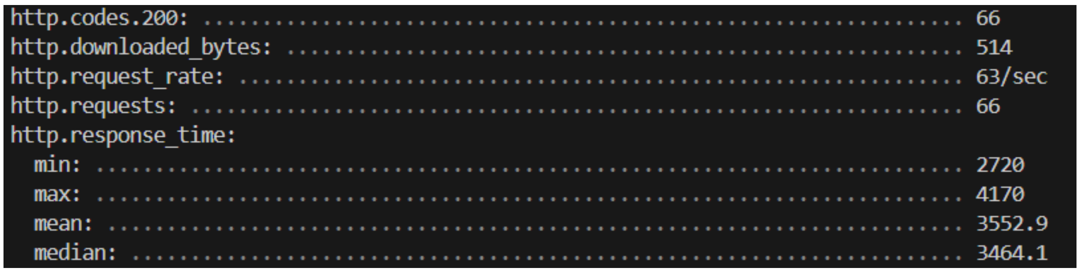
```
--------------------------------------
Metrics for period to: 13:25:30(+0000) (width: 0.999s)
--------------------------------------

http.codes.200: ............................................................ 1
http.downloaded_bytes: ..................................................... 99
http.request_rate: ......................................................... 1/sec
http.requests: ............................................................. 1
http.response_time:
  min: ..................................................................... 584
  max: ..................................................................... 584
  mean: .................................................................... 584
  median: .................................................................. 584.2
```

- PostgreSQL without cache - Delete 1 user



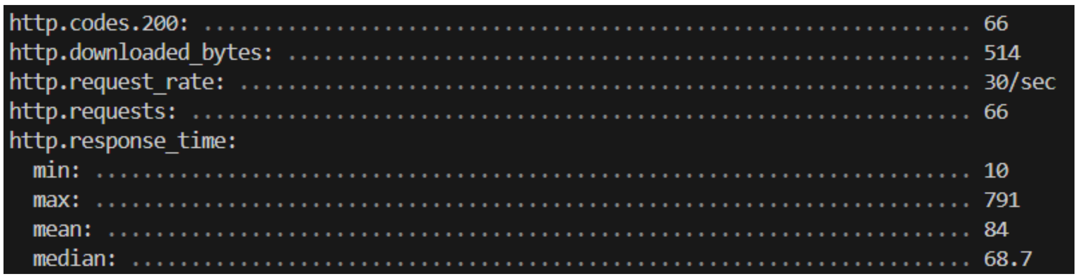
```
--------------------------------------
Metrics for period to: 16:49:30(+0000) (width: 0.432s)
--------------------------------------

http.codes.200: ............................................................ 1
http.downloaded_bytes: ..................................................... 99
http.request_rate: ......................................................... 1/sec
http.requests: ............................................................. 1
http.response_time:
  min: ..................................................................... 96
  max: ..................................................................... 96
  mean: .................................................................... 96
  median: .................................................................. 96.6
```

- NoSQL without cache - post 66 users



```
http.codes.200: ............................................................ 66
http.downloaded_bytes: ..................................................... 514
http.request_rate: ......................................................... 63/sec
http.requests: ............................................................. 66
http.response_time:
  min: ..................................................................... 2720
  max: ..................................................................... 4170
  mean: .................................................................... 3552.9
  median: .................................................................. 3464.1
```

- PostgreSQL without cache - post 66 users



```
http.codes.200: .......................................................... 66
http.downloaded_bytes: ................................................... 514
http.request_rate: ....................................................... 30/sec
http.requests: ........................................................... 66
http.response_time:
  min: ................................................................... 10
  max: ................................................................... 791
  mean: .................................................................. 84
  median: ................................................................ 68.7
```

As for the first example, we can see a huge difference in the **mean** values, with 584ms for NoSQL, and 96ms PostgreSQL. In this test we had a user with shorts, likes, and followers and had it deleted. As shown before on the screenshots of the code, and mentioned in the NoSQL and PostgreSQL explanation, the NoSQL database would have to make separate requests for each container to resolve a complex change to the database, therefore in this case it had to reach all containers, users, shorts, likes and following, one at a time. On the other hand PostgreSQL has it's advantage on being able to realize ACID transactions, and so being extremely faster than NoSQL.

For the second test, it showed a **mean** of 3552.9ms for NoSQL and 84ms for PostgreSQL. The significant speed advantage of PostgreSQL here, is due to its efficient handling of inserts, minimizing over-processing data and indexing complexity, while NoSQL in that exact matter, has an indexing functionality happening on every inserted value. Then from these examples, and more, we recognize that PostgreSQL performs the best with data that needs relational features between tables and containers (which is the case in the Tukano application).

Through these tests and understanding of both the cache and these storage systems, the best results in terms of performance and latency were achieved by using Cache and PostgreSQL.

Disclaimer :

During the implementation of the PostgreSQL, we had to change the User object table name, making it 'users'. Having this change, we also had to review our SQL queries and change the table's name on them. By mistake, on the 'searchUsers' method, we also changed the table on the NoSQL query, therefore the method won't work correctly (it should be `"SELECT * FROM User u WHERE UPPER(u.id) LIKE '%%%s%%'"`).