

# SCC Project 1 : Tukano Web App

...

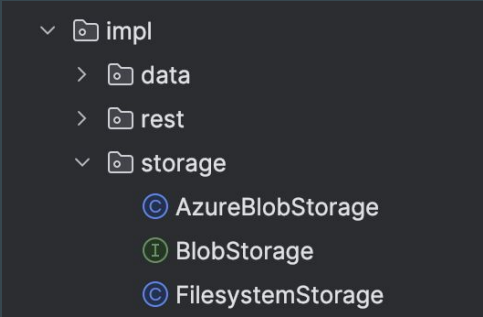
Jonas Dimitrow 71867  
Cornelius Wiehl 72009

# Overview

- Azure Blob Storage
- Azure CosmosDB
- Azure Redis (with performance Tests)
- SQL
- User Session

# Azure Blob Storage

- In order to utilize Azure Blob Storage we implemented an alternative storage class **AzureBlobStorage** that replaces the **FileSystemStorage**.
- **AzureBlobStorage** provides the necessary read, write and delete methods used by the JavaBlobs class.
- By doing so the changes to the existing structure of the project are minimal and can be easily reversed if Blob Storage is not needed or not available.



```

  v impl
    > data
    > rest
    v storage
      © AzureBlobStorage
      © BlobStorage
      © FileSystemStorage

```

A screenshot of a file explorer interface with a dark background. It shows a tree view of a project structure. The 'impl' directory is expanded, showing subdirectories 'data', 'rest', and 'storage'. The 'storage' directory is also expanded, showing three files: 'AzureBlobStorage' (with a blue copyright icon), 'BlobStorage' (with a green copyright icon), and 'FileSystemStorage' (with a blue copyright icon).

# Azure Blob Storage

- Azure Blob Storage is a suitable choice for storing large amount of unstructured data like the “shorts” used in our application.
- As we expect a regular traffic for our application the **Hot Access Tier** for our blobs is suitable.
- the blob storage configuration is written in ‘azureblob.properties’ in the project resources

# Azure Cosmos DB

- we implemented the use of azure cosmosdb with 3 classes:
  - a cosmosdb connection container that creates and holds the configured connection to cosmosdb
  - the class AzureUsers that implements the Users interface to replace the JavaUsers class
  - the class AzureShorts that implements the Shorts interface to replace the JavaShorts class
- the cosmosdb connection is placed in 'db.properties' in the project resources

```
dbtype=cosmosdb
connectionUrl=[REDACTED]
dbKey=[REDACTED]
dbName=[REDACTED]
userContainerName=u:
shortContainerName=
```

# Azure Cosmos DB - AzureUsers

- implemented as a singleton
- various null checks, logging and also password validation in its methods

```
try {
    Result<User> result = getUser(userId, pwd, useCache: true);
    if (!result.isOK()){
        Log.severe(() -> String.format("Could not find user to delete. user-id=%s\n", userId));
        return error(ErrorCode.NOT_FOUND);
    }
    User user = result.value();
    if (user == null) {
        Log.severe(() -> String.format("Could not find user to delete. user-id=%s\n", userId));
        return error(ErrorCode.NOT_FOUND);
    }
    if (!authorizationOk(user, pwd)) {
        Log.severe(() -> String.format("Wrong password for user with Id %s\n", userId));
        return error(ErrorCode.UNAUTHORIZED);
    }
}
```

- deleteUser method also deletes all user-associated shorts and the corresponding blobs

# Azure Cosmos DB - AzureShorts

- followers, likes and shorts are all documents in the same container differentiated by an additional field 'type'
  - thus for deleting a short the corresponding followers and likes can be easily retrieved and also deleted

```
try {  
    // Delete all shorts  
    String shortsQuery = "SELECT * FROM c WHERE c.type = 'SHORT' AND c.ownerId = '" + userId + "'";  
    container.queryItems(shortsQuery, new CosmosQueryRequestOptions(), Short.class)  
        .forEach(s -> container.deleteItem(s.getId(), new PartitionKey(s.getId()), new CosmosItemRequestOptions()));  
  
    // Delete all follows  
    String followsQuery = "SELECT * FROM c WHERE c.type = 'FOLLOWING' AND (c.follower = '" + userId + "' OR c.followee = '" + userId + "')";  
    container.queryItems(followsQuery, new CosmosQueryRequestOptions(), Following.class)  
        .forEach(f -> container.deleteItem(f.getId(), new PartitionKey(f.getId()), new CosmosItemRequestOptions()));  
  
    // Delete all likes  
    String likesQuery = "SELECT * FROM c WHERE c.type = 'LIKE' AND (c.userId = '" + userId + "' OR c.ownerId = '" + userId + "')";  
    container.queryItems(likesQuery, new CosmosQueryRequestOptions(), Likes.class)  
        .forEach(l -> container.deleteItem(l.getId(), new PartitionKey(l.getId()), new CosmosItemRequestOptions()));  
}
```

- also implemented as a singleton

# Azure Redis

- We choose Azure Redis Cache to improve throughput and latency for our application.
- The focus is to improve the heavily utilized method **getUser()** from our **AzureUsers** class.
- Therefore if a request is made we first look if the user is found in Cache and can be returned directly. If not a request to the database is necessary. The user is then added to the Cache to speed up future requests to that user.
- Other methods like updateUser() or deleteUser() have to be adjusted such that the cache and database are synced.



# Azure Redis

- For implementation we created a RedisCachePool class that provides a JedisPool with TLS that can be used across the application.
- To handle cache related operations we also created a CacheUtils class with appropriate methods for handling store, delete and update operations in cache.
- To avoid duplication we used a prefix based approach. Consequently we e.g. add a “user” prefix for all user related cache entries.

▼  utils

© Args


© AuthUtils

© CacheUtils

▼  impl

>  data

>  rest

>  storage

© AzureShorts

© AzureUsers

© CosmosClientContainer

© JavaBlobs

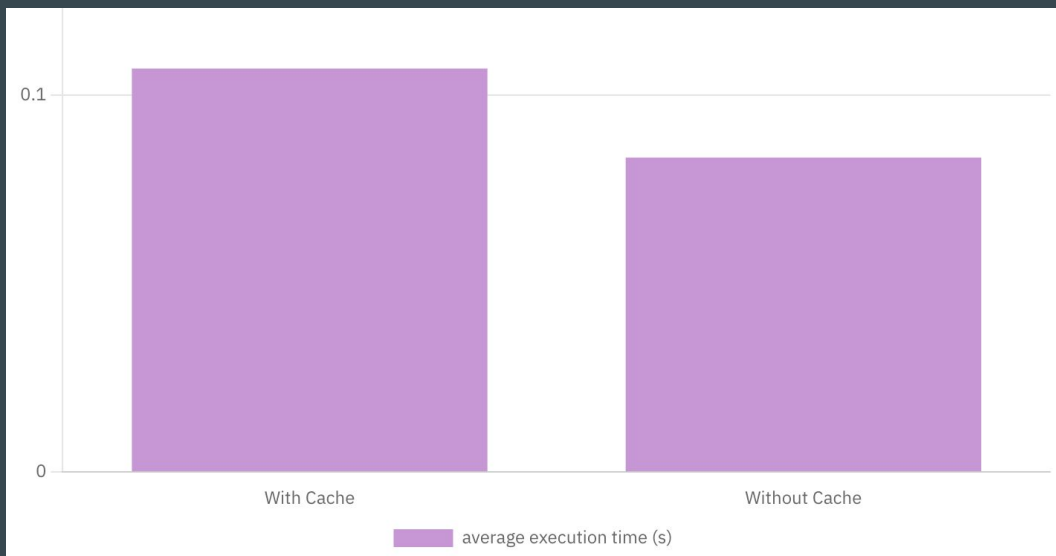
© JavaShorts

© JavaUsers

© RedisCachePool

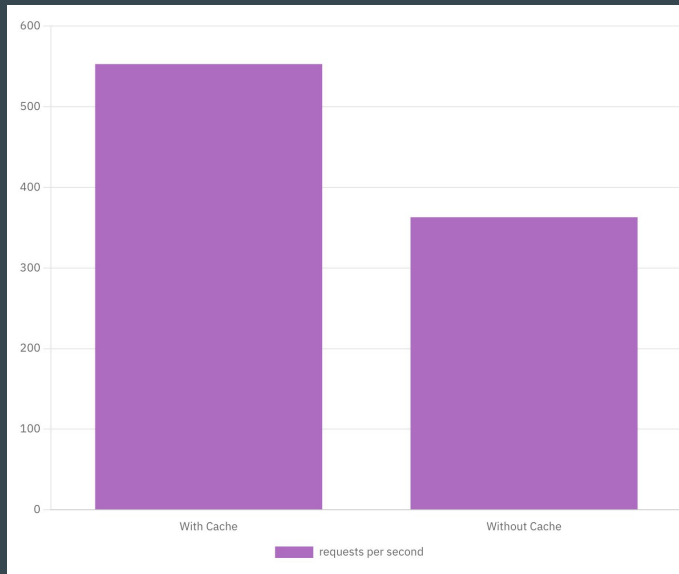
# Azure Redis Throughput Tests

- For a realistic performance test we created random users and randomly accessed the users and measure the throughput with and without the use of cache.
- The use of cache improved the overall throughput significantly.



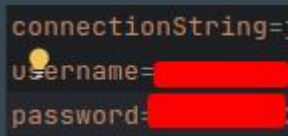
# Azure Redis Latency Tests

- For a realistic performance test we created random users and randomly accessed the users and measure the latency..
- The use of cache improved the overall latency significantly.

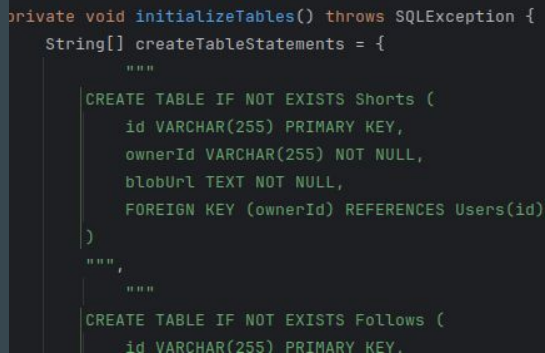


# SQL

- to enable the use of SQL databases (tested only with postgresql) we provide extra keys in the 'db.properties' file
- specifically we provided a SQL implementation for the Users and Shorts interfaces
- the biggest changes compared to the cosmosdb implementations were the queries that had to be adapted to SQL for each method as well as an initialization method for each class that creates the tables and their relations



```
connectionString=  
username=  
password=
```



```
private void initializeTables() throws SQLException {  
    String[] createTableStatements = {  
        ""  
        CREATE TABLE IF NOT EXISTS Shorts (  
            id VARCHAR(255) PRIMARY KEY,  
            ownerId VARCHAR(255) NOT NULL,  
            blobUrl TEXT NOT NULL,  
            FOREIGN KEY (ownerId) REFERENCES Users(id)  
        )  
        ,  
        ""  
        CREATE TABLE IF NOT EXISTS Follows (  
            id VARCHAR(255) PRIMARY KEY,
```

# User Session

- to imitate a user session we used to given Token class
  - when a user makes an API call for the first time (no Token exists yet) the password is validated and a new Token is created and placed in the Cache
  - for subsequent API calls our authentication routine checks if a Token was supplied and then looks it up in the Cache to authenticate the User
  - we used the password string parameter in the interfaces as double use to be either a password or a token string and we differentiate between the two by checking if the supplied string matches the format of a Token (i.e. it starts with a timestamp and theres a '-' character in the middle)

