Azure TuKano

Pedro Lopes[1][70546] and Valentim Paulo[2][70547]

[1] Nova School of Science and Technology
[2] Discipline: SCC

Abstract. This project explores the process of porting TuKano, a social network web application for short-form video sharing, to the Microsoft Azure Cloud platform. The migration aims to harness Azure's Platform as a Service (PaaS) capabilities to enhance scalability, availability, and performance. Our solution transitions TuKano's existing three-tier architecture—comprising the Users, Shorts, and Blobs services—into a robust, cloud-native implementation utilizing key Azure services. Specifically, Azure Blob Storage, Azure Cosmos DB, and Azure Cache for Redis are integrated to optimize data management and caching. Through this work, we demonstrate how cloud computing principles and Azure's service portfolio can be leveraged to develop highly efficient, distributed applications.

1 Project Development

Taking into consideration that the project bases itself on the project previously developed, take into consideration that some important information can be found in the first report developed.

1.1 Implementation

In order to deploy the the adapted version of Tukano, using Docker and Kubernetes, we had to resort to the cointainerization of the services previously used:

- **PostgreSQL:** This is a relational database management system that will be responsible for storing all structured data from the project, such as user information, metadata, and other critical information. PostgreSQL offers advanced features like ACID compliance, strong data integrity, and powerful querying capabilities. It is highly extensible, supports complex data structures, and provides robust performance for both small and large-scale applications. This makes it a reliable choice for projects requiring structured data management with flexibility and scalability.
- **Blob Storage:** This is also a storage service, that will be responsible for storing all the unstructured data from the project like videos or images, but in our case, will save the shorts. Just like Cosmos DB this service provides secure and scalable storage for object data that can be quickly accessed by different applications.

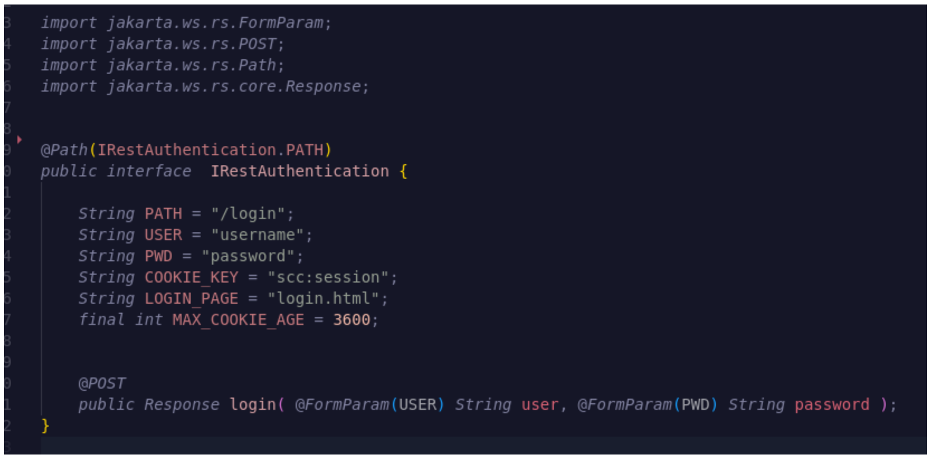
- **Cache for Redis:** This is a managed cache service based on Redis, which is an open source in-memory storage system often used to improve application performance by temporarily storing data in memory, rather than always accessing the database or other persistent data source. Caching is essential for improving performance and reducing latency.

1.2 Authentication

In the context of the project, implementing user session authentication ensures that only authorized users can interact with the Blob Service, safeguarding data integrity and security. The authentication mechanism we developed, works by logging in the user. When successfully logged in, the authentication returns a cookie that contains a Session ID, which is stored in the Cache Redis, and depending on the configuration, stays available for a certain time (in our solution Session ID's are available for 1 hour).

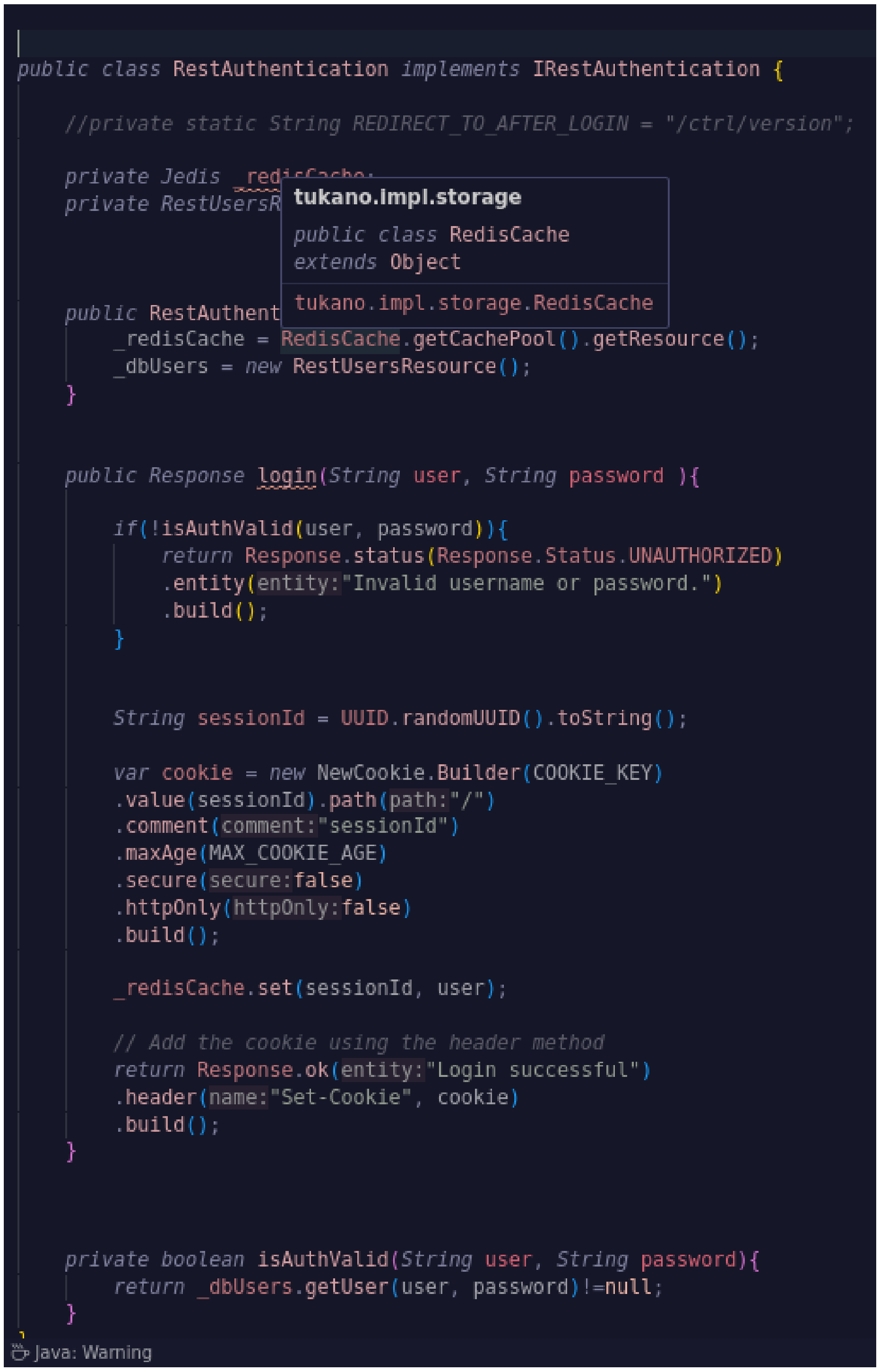
The Session ID token acts as a gatekeeper for all operations involving the Blob Service. If a Session ID is not valid in the Cache Redis, users are prevented from performing any actions related to the blobs. This design enforces strict access control, ensuring that resources are only accessible to authenticated users. By leveraging Cache Redis for Session ID's storage, the system balances security with efficiency, enabling rapid verification of Session ID's while adhering to best practices in session management and data protection.

The following images, show all the code that we had to develop in order to successfully implement the authentication:



```
import jakarta.ws.rs.FormParam;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.core.Response;


@Path(IRestAuthentication.PATH)
public interface  IRestAuthentication {

    String PATH = "/login";
    String USER = "username";
    String PWD = "password";
    String COOKIE_KEY = "scc:session";
    String LOGIN_PAGE = "login.html";
    final int MAX_COOKIE_AGE = 3600;


    @POST
    public Response login( @FormParam(USER) String user, @FormParam(PWD) String password );
}
```

Fig. 1. Authentication endpoint

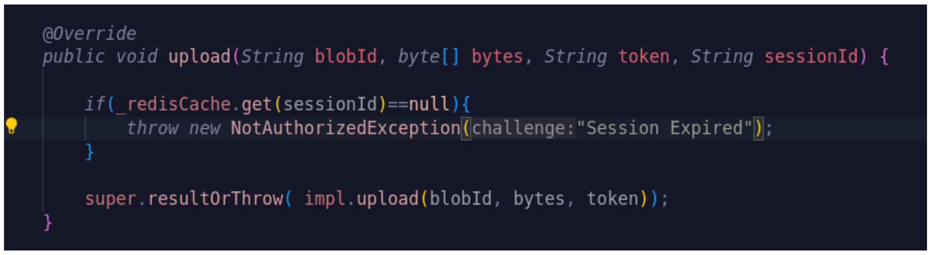


```
public class RestAuthentication implements IRestAuthentication {

    //private static String REDIRECT_TO_AFTER_LOGIN = "/ctrl/version";

    private Jedis _redisCache;
    private RestUsersR

    public RestAuthent
        _redisCache = RedisCache.getCachePool().getResource();
        _dbUsers = new RestUsersResource();
    }

    public Response login(String user, String password ){

        if(!isAuthValid(user, password)){
            return Response.status(Response.Status.UNAUTHORIZED)
            .entity(entity:"Invalid username or password.")
            .build();
        }

        String sessionId = UUID.randomUUID().toString();

        var cookie = new NewCookie.Builder(COOKIE_KEY)
        .value(sessionId).path(path:"/")
        .comment(comment:"sessionId")
        .maxAge(MAX_COOKIE_AGE)
        .secure(secure:false)
        .httpOnly(httpOnly:false)
        .build();

        _redisCache.set(sessionId, user);

        // Add the cookie using the header method
        return Response.ok(entity:"Login successful")
        .header(name:"Set-Cookie", cookie)
        .build();
    }

    private boolean isAuthValid(String user, String password){
        return _dbUsers.getUser(user, password)!=null;
    }
```

```
tukano.impl.storage

public class RedisCache
extends Object

tukano.impl.storage.RedisCache
```

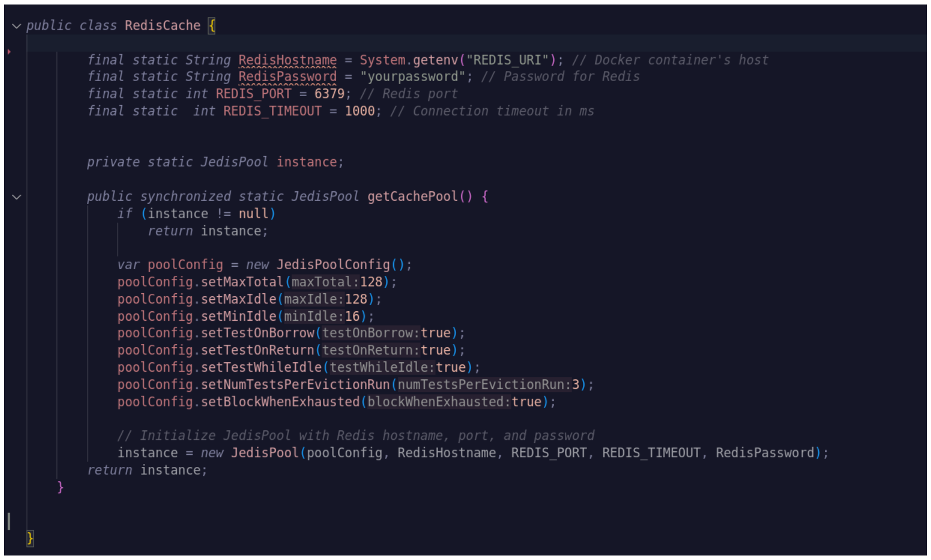
Java: Warning

Fig. 2. Authentication



```java
@Override
public void upload(String blobId, byte[] bytes, String token, String sessionId) {

    if(_redisCache.get(sessionId)==null){
        throw new NotAuthorizedException(challenge:"Session Expired");
    }

    super.resultOrThrow( impl.upload(blobId, bytes, token));
}
```

Fig. 3. Example of a Blob action



```java
public class RedisCache {

        final static String RedisHostname = System.getenv("REDIS_URI"); // Docker container's host
        final static String RedisPassword = "yourpassword"; // Password for Redis
        final static int REDIS_PORT = 6379; // Redis port
        final static  int REDIS_TIMEOUT = 1000; // Connection timeout in ms


        private static JedisPool instance;

        public synchronized static JedisPool getCachePool() {
            if (instance != null)
                return instance;

            var poolConfig = new JedisPoolConfig();
            poolConfig.setMaxTotal(maxTotal:128);
            poolConfig.setMaxIdle(maxIdle:128);
            poolConfig.setMinIdle(minIdle:16);
            poolConfig.setTestOnBorrow(testOnBorrow:true);
            poolConfig.setTestOnReturn(testOnReturn:true);
            poolConfig.setTestWhileIdle(testWhileIdle:true);
            poolConfig.setNumTestsPerEvictionRun(numTestsPerEvictionRun:3);
            poolConfig.setBlockWhenExhausted(blockWhenExhausted:true);

            // Initialize JedisPool with Redis hostname, port, and password
            instance = new JedisPool(poolConfig, RedisHostname, REDIS_PORT, REDIS_TIMEOUT, RedisPassword);
        return instance;
    }


}
```

Fig. 4. Cache Redis client

1.3 Containerization

In order to successfully deploy the Tukano using kubernetes, we needed to containerize both Web App and the services. By doing so, we developed a architecture where, the Web App container, being the main container, would be able to access all other containers in order to make any type of needed action. The following image, shows what this architecture looks like.

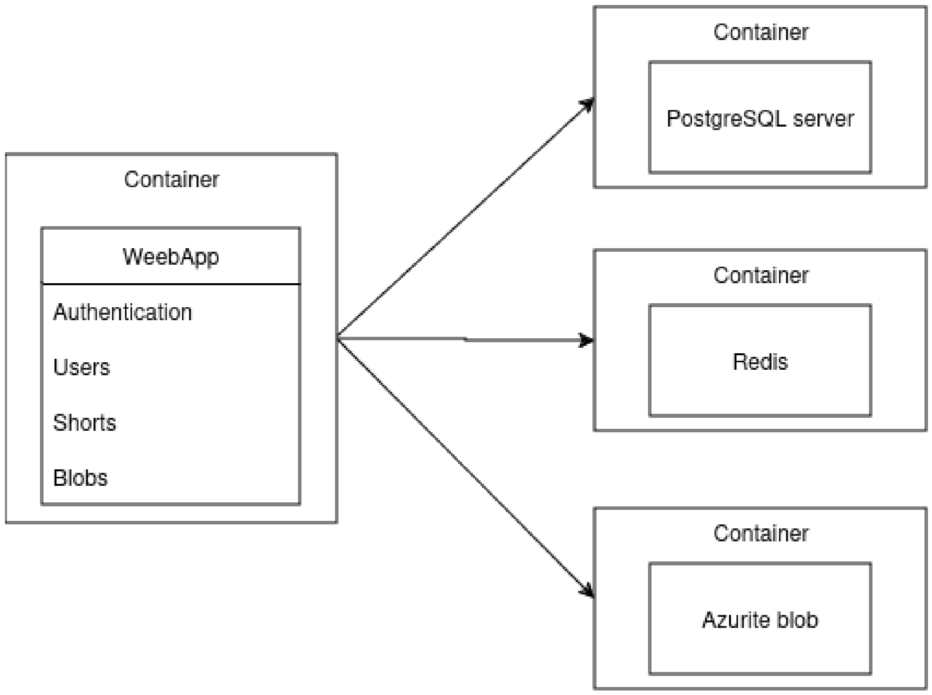


Fig. 5. Containerized Solution Architecture

Services

- **PostgreSQL:** In order to containerize the PostgreSQL service we used the official PostgreSQL Docker image, available at Docker Hub.
- **Cache Redis:** In order to containerize the Redis service we used the official Redis Docker image (redis/redis-stack-server), available at Docker Hub.
- **Blob Storage:** In order to containerize the Azurite we used the official Microsoft Azurite Docker image (mcr.microsoft.com/azure-storage/azurite azurite-blob) whit a use of a persistent volume to store the data, available at Docker Hub.

Web App

In order to containerize the Java web app we needed to take several steps:

- **Get the .war:** We compiled the project utilizing maven, on the target directory is the .war file.
- **Hibernate.cfg.xlm:** We configured the Hibernate.cfg.xlm whit the connection url, user name and password to the containerized PostgreSQL server, shown in the following image.

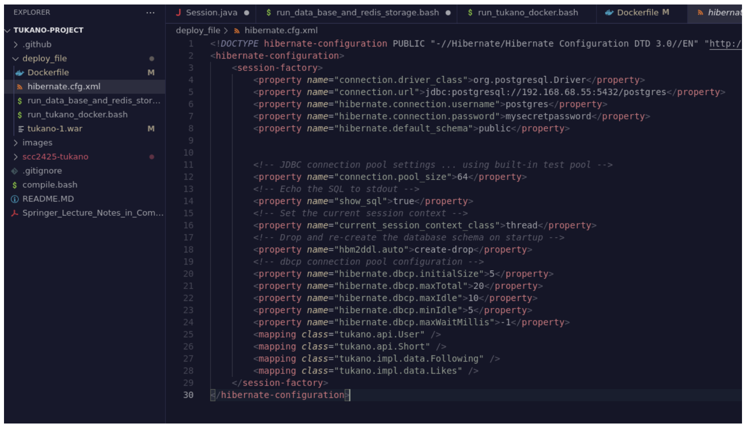

```
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http:/
<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">org.postgresql.Driver</property>
        <property name="connection.url">jdbc:postgresql://192.168.68.55:5432/postgres</property>
        <property name="hibernate.connection.username">postgres</property>
        <property name="hibernate.connection.password">mysecretpassword</property>
        <property name="hibernate.default_schema">public</property>


        <!-- JDBC connection pool settings ... using built-in test pool -->
        <property name="connection.pool_size">64</property>
        <!-- Echo the SQL to stdout -->
        <property name="show_sql">true</property>
        <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>
        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create-drop</property>
        <!-- dbcp connection pool configuration -->
        <property name="hibernate.dbcp.initialSize">5</property>
        <property name="hibernate.dbcp.maxTotal">20</property>
        <property name="hibernate.dbcp.maxIdle">10</property>
        <property name="hibernate.dbcp.minIdle">5</property>
        <property name="hibernate.dbcp.maxWaitMillis">-1</property>
        <mapping class="tukano.api.User" />
        <mapping class="tukano.api.Short" />
        <mapping class="tukano.impl.data.Following" />
        <mapping class="tukano.impl.data.Likes" />
    </session-factory>
</hibernate-configuration>
```

Fig. 6. Hibernater.cfg.xlm

- **Create the docker file:** Having the .war file and the hibernate all in the same directory, we created a Dockerfile and configurated it in order to allow the creation of the Docker image. The configuration can be seen in the following image.

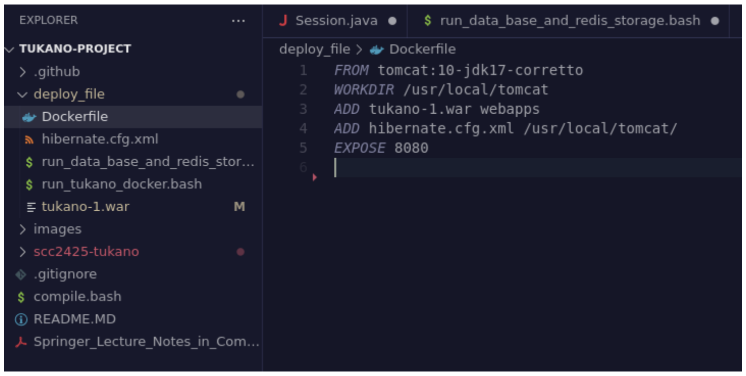

```
FROM tomcat:10-jdk17-corretto
WORKDIR /usr/local/tomcat
ADD tukano-1.war webapps
ADD hibernate.cfg.xml /usr/local/tomcat/
EXPOSE 8080
```

Fig. 7. Docker File

- **Build the image:** After configurating it, use the following command in order to create and run the container:

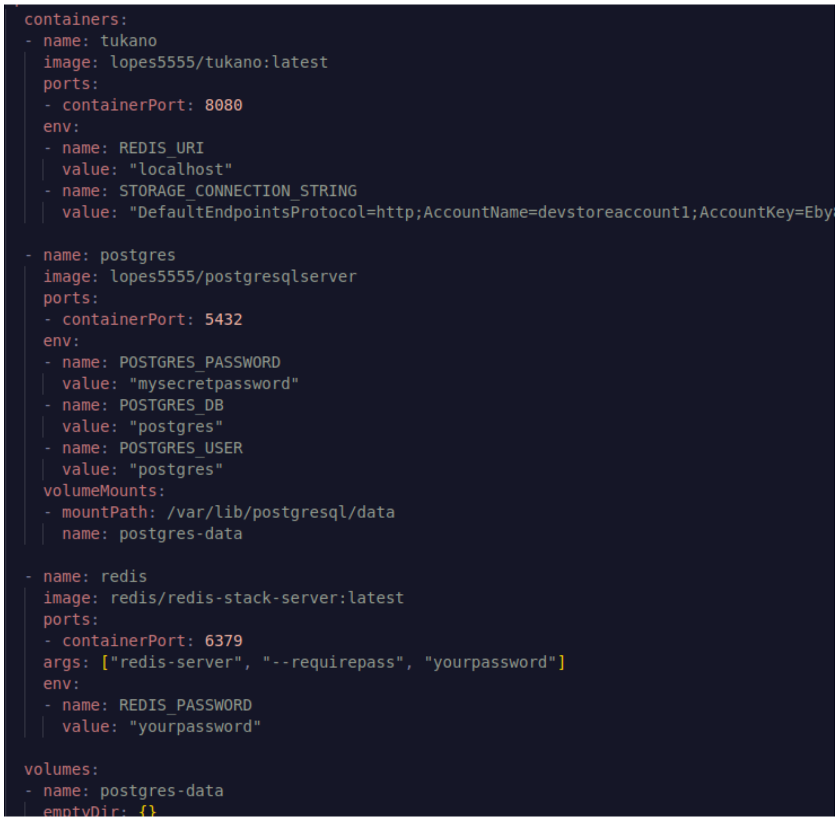
```
docker run -d -e REDIS_URI="redis_url" -e
    STORAGE_CONNECTION_STRING="connection_string" --
    net=host tukano
```

2 Project Deployment to Kubernets

In order to successfully deploy the solution, we had to publish the Docker images to Docker Hub. This was a crucial move to ensure that the application's containerized environments were accessible.

After publishing in order to make the deploy using Kubernetes, we needed to create a '.yaml' file containing all the configurations (like setting up the environment variables) required for the deployment. This file defined which containers would be part of the solution, including the Tukano web app, Redis, PostgreSQL and Blob Storage.

The following image shows the configuration of the file, taking into consideration that this specific configuration deployed all the containers in the same Pod, allowing each container to comunicate via localhost.



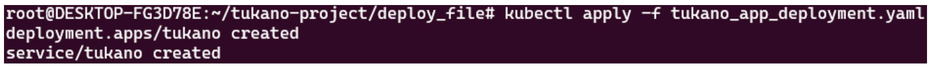
```yaml
containers:
- name: tukano
  image: lopes5555/tukano:latest
  ports:
  - containerPort: 8080
  env:
  - name: REDIS_URI
    value: "localhost"
  - name: STORAGE_CONNECTION_STRING
    value: "DefaultEndpointsProtocol=http;AccountName=devstoreaccount1;AccountKey=Eby

- name: postgres
  image: lopes5555/postgresqlserver
  ports:
  - containerPort: 5432
  env:
  - name: POSTGRES_PASSWORD
    value: "mysecretpassword"
  - name: POSTGRES_DB
    value: "postgres"
  - name: POSTGRES_USER
    value: "postgres"
  volumeMounts:
  - mountPath: /var/lib/postgresql/data
    name: postgres-data

- name: redis
  image: redis/redis-stack-server:latest
  ports:
  - containerPort: 6379
  args: ["redis-server", "--requirepass", "yourpassword"]
  env:
  - name: REDIS_PASSWORD
    value: "yourpassword"

volumes:
- name: postgres-data
  emptyDir: {}
```

Fig. 8. .yaml file

2.1 Deployment Process

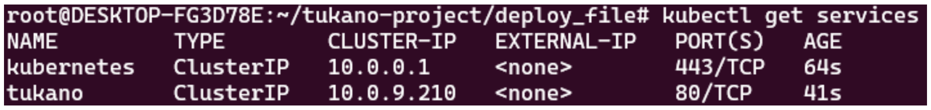
The deployment process consists in the execution of multiple commands, in order to make it easier to understand we have several images that show the process, including the commands for the deployment, as well as the commands of some tests that were made.

Important. Take into consideration that for the deployment to be successful, there needs to be created a resource group, a service principal and a cluster, being all these associated to the microsoft azure account. Dont forget all these are monetized services.



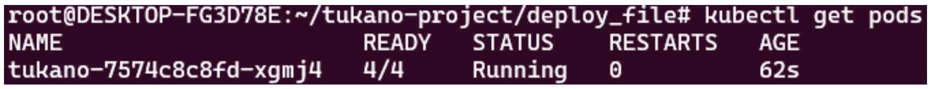
```
root@DESKTOP-FG3D78E:~/tukano-project/deploy_file# kubectl apply -f tukano_app_deployment.yaml
deployment.apps/tukano created
service/tukano created
```

Fig. 9. Deploying accordingly the .yaml file



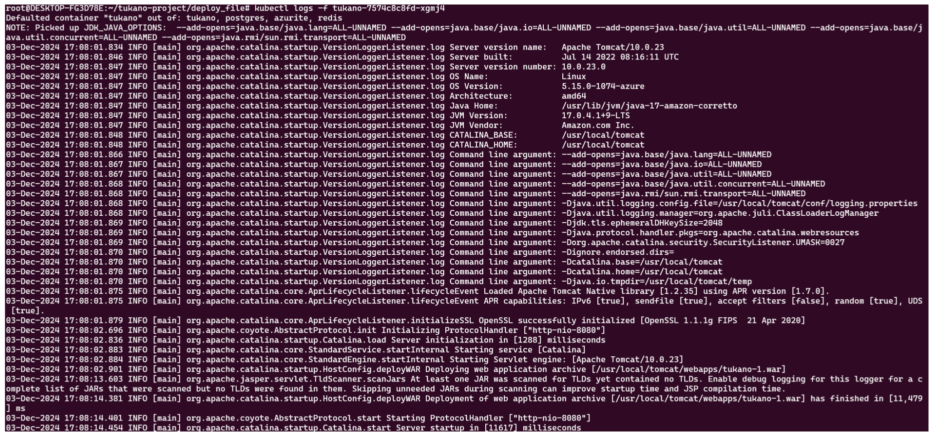
```
root@DESKTOP-FG3D78E:~/tukano-project/deploy_file# kubectl get services
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.0.0.1      <none>        443/TCP   64s
tukano       ClusterIP   10.0.9.210    <none>        80/TCP    41s
```

Fig. 10. Check if the service was created



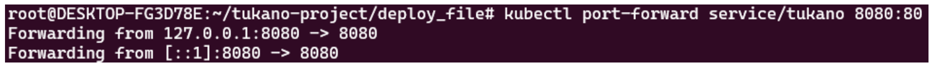
```
root@DESKTOP-FG3D78E:~/tukano-project/deploy_file# kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
tukano-7574c8c8fd-xgmj4     4/4     Running   0          62s
```

Fig. 11. Check the pod inside de service



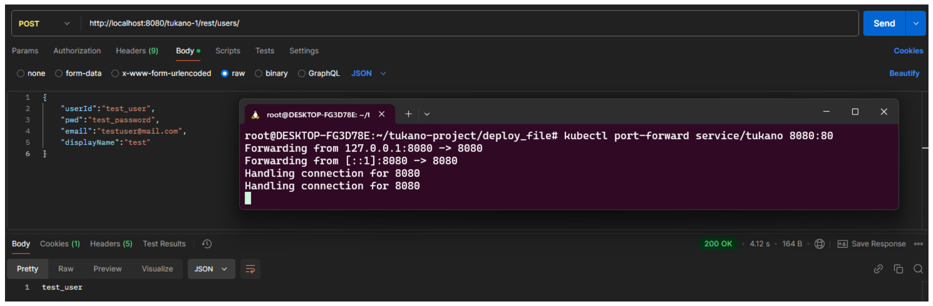
```
root@DESKTOP-FG3D78E:~/tukano-project/deploy_file# kubectl logs -f tukano-7574c8c8fd-xgmj4
Defaulted container "tukano" out of: tukano, postgres, azurite, redis
NOTE: Picked up JDK_JAVA_OPTIONS:  --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.io=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED --add-opens=java.base/j
ava.util.concurrent=ALL-UNNAMED --add-opens=java.rmi/sun.rmi.transport=ALL-UNNAMED
03-Dec-2024 17:08:01.834 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version name:   Apache Tomcat/10.0.23
03-Dec-2024 17:08:01.846 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server built:          Jul 14 2022 08:16:11 UTC
03-Dec-2024 17:08:01.847 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Server version number: 10.0.23.0
03-Dec-2024 17:08:01.847 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Name:               Linux
03-Dec-2024 17:08:01.847 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log OS Version:            5.15.0-1074-azure
03-Dec-2024 17:08:01.847 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Architecture:          amd64
03-Dec-2024 17:08:01.847 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Java Home:             /usr/lib/jvm/java-17-amazon-corretto
03-Dec-2024 17:08:01.847 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log JVM Version:           17.0.4.1+9-LTS
03-Dec-2024 17:08:01.847 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log JVM Vendor:            Amazon.com Inc.
03-Dec-2024 17:08:01.848 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log CATALINA_BASE:         /usr/local/tomcat
03-Dec-2024 17:08:01.848 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log CATALINA_HOME:         /usr/local/tomcat
03-Dec-2024 17:08:01.866 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-opens=java.base/java.lang=ALL-UNNAMED
03-Dec-2024 17:08:01.867 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-opens=java.base/java.io=ALL-UNNAMED
03-Dec-2024 17:08:01.867 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-opens=java.base/java.util=ALL-UNNAMED
03-Dec-2024 17:08:01.868 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-opens=java.base/java.util.concurrent=ALL-UNNAMED
03-Dec-2024 17:08:01.868 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: --add-opens=java.rmi/sun.rmi.transport=ALL-UNNAMED
03-Dec-2024 17:08:01.868 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djava.util.logging.config.file=/usr/local/tomcat/conf/logging.properties
03-Dec-2024 17:08:01.868 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
03-Dec-2024 17:08:01.869 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djdk.tls.ephemeralDHKeySize=2048
03-Dec-2024 17:08:01.869 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djava.protocol.handler.pkgs=org.apache.catalina.webresources
03-Dec-2024 17:08:01.869 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dorg.apache.catalina.security.SecurityListener.UMASK=0027
03-Dec-2024 17:08:01.870 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dignore.endorsed.dirs=
03-Dec-2024 17:08:01.870 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dcatalina.base=/usr/local/tomcat
03-Dec-2024 17:08:01.870 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Dcatalina.home=/usr/local/tomcat
03-Dec-2024 17:08:01.870 INFO [main] org.apache.catalina.startup.VersionLoggerListener.log Command line argument: -Djava.io.tmpdir=/usr/local/tomcat/temp
03-Dec-2024 17:08:01.875 INFO [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent Loaded Apache Tomcat Native library [1.2.35] using APR version [1.7.0].
03-Dec-2024 17:08:01.875 INFO [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent APR capabilities: IPv6 [true], sendfile [true], accept filters [false], random [true], UDS
 [true].
03-Dec-2024 17:08:01.879 INFO [main] org.apache.catalina.core.AprLifecycleListener.initializeSSL OpenSSL successfully initialized [OpenSSL 1.1.1g FIPS  21 Apr 2020]
03-Dec-2024 17:08:02.696 INFO [main] org.apache.coyote.AbstractProtocol.init Initializing ProtocolHandler ["http-nio-8080"]
03-Dec-2024 17:08:02.836 INFO [main] org.apache.catalina.startup.Catalina.load Server initialization in [1288] milliseconds
03-Dec-2024 17:08:02.883 INFO [main] org.apache.catalina.core.StandardService.startInternal Starting service [Catalina]
03-Dec-2024 17:08:02.884 INFO [main] org.apache.catalina.core.StandardEngine.startInternal Starting Servlet engine: [Apache Tomcat/10.0.23]
03-Dec-2024 17:08:02.901 INFO [main] org.apache.catalina.startup.HostConfig.deployWAR Deploying web application archive [/usr/local/tomcat/webapps/tukano-1.war]
03-Dec-2024 17:08:13.603 INFO [main] org.apache.jasper.servlet.TldScanner.scanJars At least one JAR was scanned for TLDs yet contained no TLDs. Enable debug logging for this logger for a c
omplete list of JARs that were scanned but no TLDs were found in them. Skipping unneeded JARs during scanning can improve startup time and JSP compilation time.
03-Dec-2024 17:08:14.381 INFO [main] org.apache.catalina.startup.HostConfig.deployWAR Deployment of web application archive [/usr/local/tomcat/webapps/tukano-1.war] has finished in [11,479
] ms
03-Dec-2024 17:08:14.401 INFO [main] org.apache.coyote.AbstractProtocol.start Starting ProtocolHandler ["http-nio-8080"]
03-Dec-2024 17:08:14.454 INFO [main] org.apache.catalina.startup.Catalina.start Server startup in [11617] milliseconds
```

Fig. 12. Check the logs of the Web App



```
root@DESKTOP-FG3D78E:~/tukano-project/deploy_file# kubectl port-forward service/tukano 8080:80
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

Fig. 13. Activating forwarding to make tests



```
http://localhost:8080/tukano-1/rest/users/
```

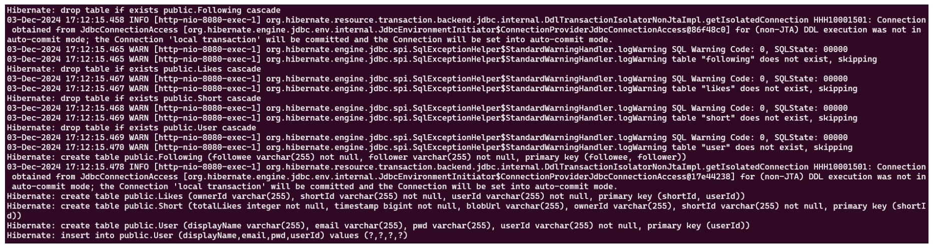
```
{
    "userId":"test_user",
    "pwd":"test_password",
    "email":"testuser@mail.com",
    "displayName":"test"
}
```

```
root@DESKTOP-FG3D78E:~/tukano-project/deploy_file# kubectl port-forward service/tukano 8080:80
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
Handling connection for 8080
Handling connection for 8080
```

200 OK · 4.12 s · 164 B

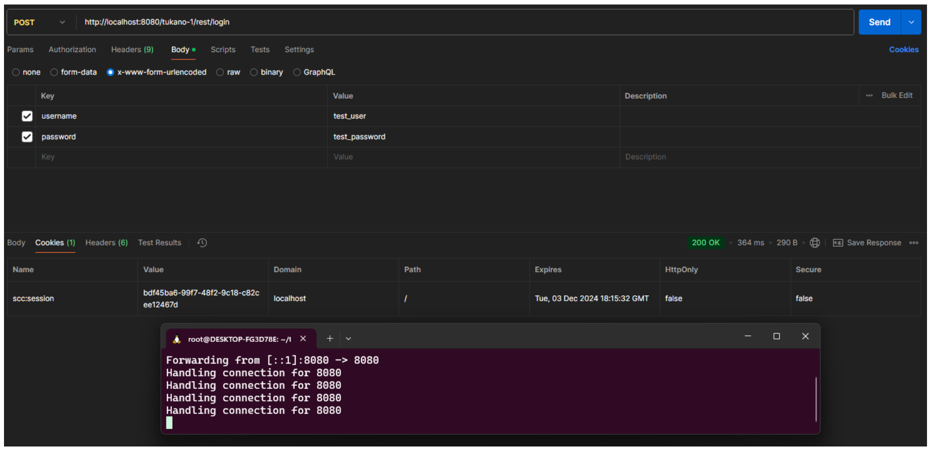
```
test_user
```

Fig. 14. Testing create user with Postman



```
Hibernate: drop table if exists public.Following cascade
03-Dec-2024 17:12:15.458 INFO [http-nio-8080-exec-1] org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl.getIsolatedConnection HHH10001501: Connection
 obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJdbcConnectionAccess@86f48c0] for (non-JTA) DDL execution was not in
auto-commit mode; the Connection 'local transaction' will be committed and the Connection will be set into auto-commit mode.
03-Dec-2024 17:12:15.465 WARN [http-nio-8080-exec-1] org.hibernate.engine.jdbc.spi.SqlExceptionHelper$StandardWarningHandler.logWarning SQL Warning Code: 0, SQLState: 00000
03-Dec-2024 17:12:15.465 WARN [http-nio-8080-exec-1] org.hibernate.engine.jdbc.spi.SqlExceptionHelper$StandardWarningHandler.logWarning table "following" does not exist, skipping
Hibernate: drop table if exists public.Likes cascade
03-Dec-2024 17:12:15.467 WARN [http-nio-8080-exec-1] org.hibernate.engine.jdbc.spi.SqlExceptionHelper$StandardWarningHandler.logWarning SQL Warning Code: 0, SQLState: 00000
03-Dec-2024 17:12:15.467 WARN [http-nio-8080-exec-1] org.hibernate.engine.jdbc.spi.SqlExceptionHelper$StandardWarningHandler.logWarning table "likes" does not exist, skipping
Hibernate: drop table if exists public.Short cascade
03-Dec-2024 17:12:15.468 WARN [http-nio-8080-exec-1] org.hibernate.engine.jdbc.spi.SqlExceptionHelper$StandardWarningHandler.logWarning SQL Warning Code: 0, SQLState: 00000
03-Dec-2024 17:12:15.469 WARN [http-nio-8080-exec-1] org.hibernate.engine.jdbc.spi.SqlExceptionHelper$StandardWarningHandler.logWarning table "short" does not exist, skipping
Hibernate: drop table if exists public.User cascade
03-Dec-2024 17:12:15.469 WARN [http-nio-8080-exec-1] org.hibernate.engine.jdbc.spi.SqlExceptionHelper$StandardWarningHandler.logWarning SQL Warning Code: 0, SQLState: 00000
03-Dec-2024 17:12:15.470 WARN [http-nio-8080-exec-1] org.hibernate.engine.jdbc.spi.SqlExceptionHelper$StandardWarningHandler.logWarning table "user" does not exist, skipping
Hibernate: create table public.Following (followee varchar(255) not null, follower varchar(255) not null, primary key (followee, follower))
03-Dec-2024 17:12:15.478 INFO [http-nio-8080-exec-1] org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl.getIsolatedConnection HHH10001501: Connection
 obtained from JdbcConnectionAccess [org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJdbcConnectionAccess@17e44238] for (non-JTA) DDL execution was not in
 auto-commit mode; the Connection 'local transaction' will be committed and the Connection will be set into auto-commit mode.
Hibernate: create table public.Likes (ownerId varchar(255), shortId varchar(255) not null, userId varchar(255) not null, primary key (shortId, userId))
Hibernate: create table public.Short (totalLikes integer not null, timestamp bigint not null, blobUrl varchar(255), ownerId varchar(255), shortId varchar(255) not null, primary key (shortI
d))
Hibernate: create table public.User (displayName varchar(255), email varchar(255), pwd varchar(255), userId varchar(255) not null, primary key (userId))
Hibernate: insert into public.User (displayName,email,pwd,userId) values (?,?,?,?)
```

Fig. 15. Cheking server logs



POST http://localhost:8080/tukano-1/rest/login

none | form-data | [✓ x-www-form-urlencoded] | raw | binary | GraphQL

	Key	Value	Description
✓	username	test_user	
✓	password	test_password	
	Key	Value	Description

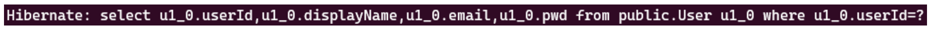
200 OK · 364 ms · 290 B

Name	Value	Domain	Path	Expires	HttpOnly	Secure
scc:session	bdf45ba6-99f7-48f2-9c18-c82c ee12467d	localhost	/	Tue, 03 Dec 2024 18:15:32 GMT	false	false

root@DESKTOP-FG3D78E: ~/t

```
Forwarding from [::1]:8080 -> 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
```

Fig. 16. checking login endpoint whit postman



```
Hibernate: select u1_0.userId,u1_0.displayName,u1_0.email,u1_0.pwd from public.User u1_0 where u1_0.userId=?
```

Fig. 17. checking the server logs

3 Execution guide

When the implementation was finished, we decided to develop a guide that allows anyone to create and test the solution locally in their respective machine. This guide will be responsible for creating 4 docker containers, one for each component mentioned before in the implementation. In order to visualize the guide, click the following URL. This will take you to the project's github repository where if you scroll down you can find a collapsible section for the Project part 2, where you can check the steps needed.

- https://github.com/pedroLopes5555/tukano-project

4 Conclusions

This project explored the practical application of Docker and Kubernetes, providing hands-on experience in containerizing microservices, deploying them on Azure, and orchestrating them using Kubernetes. It emphasized the transition from traditional deployment methods to modern containerized and cloud-based solutions, highlighting their scalability, efficiency, and versatility for developing and managing applications in distributed environments.