VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Nguyen Dang Toan

Building a Caching Module in a Cloud-Based System

School of Technology
2022

ACKNOWLEDGMENTS

First of all, I would like to thank all my teachers at VAMK and especially Dr. Ghodrat Moghadampour - my supervisor of this thesis project, you have always supported me in the process of my thesis writing. Moreover, the knowledge I have gained from Mr. Moghadampour during my studies at VAMK is very valuable and memorable, which will be the lessons I will apply in the future.

Next, I want to thank Mr. Quang, who has helped with the English grammar.

Finally, I am grateful to my parents. They are in Vietnam, but their love has been an indispensable source of motivation throughout my studies.

Nguyen Dang Tri Toan
Vaasa, Finland
20.11.2021

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Nguyen Dang Toan
Title	Building a Caching Module in a Cloud-Based System
Year	2021
Language	English
Pages	38 + 10 Appendices
Name of Supervisor	Ghodrat Moghadampour

The topic of the thesis was to develop a cloud-based application running on the Amazon Web Services platform which was responsible for caching and validating data.

The background was that under certain circumstances of online payment processes, the data must be preserved for a short period of time and storing them onto database was not a wise choice due to the latency and heavy workload, therefore caching was a good solution. In addition, the application was required to be light, reliable and easy to manage as defined by the concept of microservices. For this purpose, various technologies were used; Quarkus, Docker container, Redis and AWS.

The main objectives of the thesis were to develop a module for online payment processes which will validate and cache payment information as well as to provide a useful guideline for those who are interested in how to build a compact serverless application which can be used by any AWS-based software systems.

This application has fully achieved its goals. By using the latest powerful technologies, such as Quarkus, Docker, and Amazon Web Services, the application is lighter and faster than expected and making future maintenance and enhancement will be easier.

Keywords AWS cloud service, serverless, Quarkus, Redis and Docker

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

LIST OF CODE SNIPPETS

LIST OF ABBREVIATIONS

AWS – Amazon Web Services

ECR – Elastic Container Registry

ECS – Elastic Container Service

CLI – Command Line Interface

API - Application Programming Interface

RDMS - Relational Database Management System

1 INTRODUCTION

In the field of software development, system expansion and upgrades are inevitable. As a result, the system gradually becomes complicated and difficult to maintain. Therefore, many cloud services have been created and they provide companies many applications to focus on their own business without spending too many resources on managing the underlying infrastructure. This thesis presents one of the ways to help businesses that is Serverless. Businesses do not have allocate a lot of resources to manage their own server. There are multiple advantages:

- Compared with traditional server computing, it saves up to 70% of the cost and users do not need to pay for additional services. /1/
- Simple to manage with a small team and no need to deal with physical server. /1/
- Serverless application offers a great scalable solution and provides needed resource immediately. The service can be scaled up and down instantly when needed. /1/
- Cloud providers also support many more services that users can flexibly choose to implement together with their systems. /1/

This application is hosted on a serverless, which is responsible for storing the payment forms sent by the user to the cache.

This introduction will include the background and the description as well as the objectives of this thesis.

1.1 Background and Description

Caching is a technique that is used and applied in many systems in general and applications in particular. Typical examples are those applications that need faster data access, or the ones that need to store data only for a short period of time. The online payment system is no exception.

The online payment process starts when an unauthenticated user posts a payment request to this application. The requested data is then validated and cached while waiting for the user to authenticating. After the user completes the authentication, the data is fetched from the cache to start an actual payment process. So that, data need to be stored just during the time users authenticate themselves.

This application is a small module that is consumed by a larger software application.

1.2 Objectives

The purpose of this thesis is to document the process of creating a stateless cloud service and how to use advanced technology to build a lightweight and fast application.

The thesis consists of six sections. The first part is an introduction to the project and thesis. It states the goal that the research is set to achieve and explains the research scenario. The second part reviews relevant technologies and tools used to build the application. The third part contains a detailed description of the project, as well as its specifications and diagrams. The fourth part reports the process of building, deploying the applications. The fifth one covers the testing method that was used to test the flow of production. The last part presents a drawn conclusion and how this application can be further improved more in the future.

2 TECHNOLOGICAL BACKGROUND

This section introduces the related technology stack used in this application. Since the application is considered a cloud-based serverless service, many AWS stacks were involved. Quarkus was the main actor here, which is a framework that is used to develop Java application. This application used Redis - the most powerful memory data structure storage at the present. Docker was used to build this application as a container. Finally, Robot Framework, was used for testing purposes.

2.1 Cloud Services

To achieve the serverless architecture, the first of the things is choosing the right Cloud Service that fits the needs of the application, as well as easy for a business to operate smoothly. There are multiple Cloud Service Providers that can provide a reliable, flexible infrastructure, such as AWS, Azure, and Google Cloud Platform.

AWS was chosen for this application due to its most powerful platform.

2.1.1 AWS

AWS stands for Amazon Web Services. It offers a variety of features and services, such as serverless computing, application services, storage options and databases. AWS allows developers to build their applications without worrying about managing the underlying infrastructure. This application uses several services that are offered by AWS such as ECR, ECS, CloudFormation, Secrets Manager. /2/

2.1.2 Elastic Container Registry (ECR)

Elastic Container Registry (ECR) is a repository for sharing, storing, deploying container images and artifacts, such as Docker image. ECR also works with Docker CLI, so that we can pull or push images from our development environments to ECR. /3/

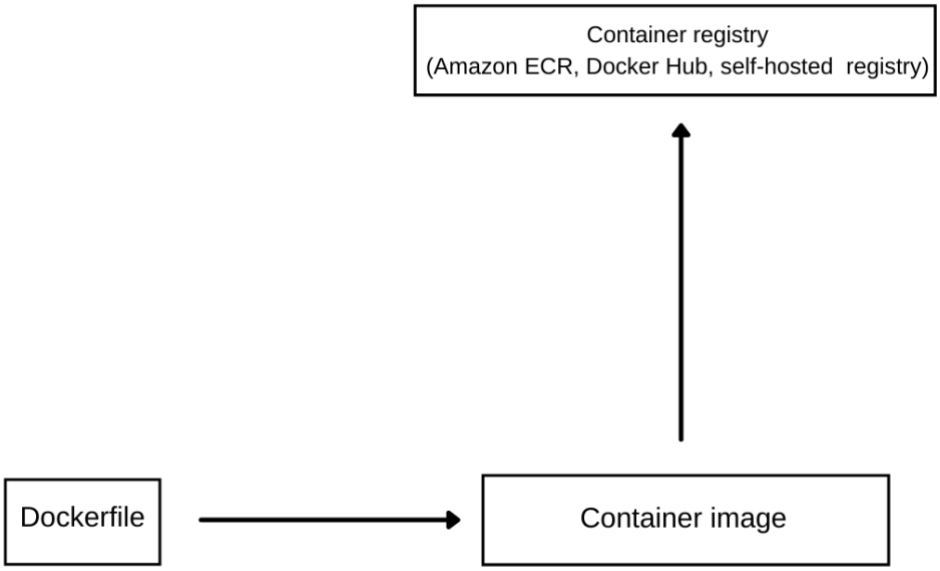

Figure 1. ECR in the execution process of System using Docker. /3/

ECR is needed when working with containers, it allows users to operate their own storage without touching the infrastructure or worrying about the scalability. It offers users some features such as cleaning up unused images, scanning vulnerabilities on images, or sharing cross region and account. /3/

2.1.3 Elastic Container Service (ECS)

"Amazon Elastic Container Service (ECS) is a highly scalable, high-performance container management service that makes it easy to run, stop, and manage containers on a cluster". /4/

It can be used to host a serverless infrastructure by running a service or task using Fargate launch type, or by using EC2 launch type to run instances. /5/

To get containers spin up, users must define their own Task Definition. The Task Definition can be defined within a text file in json or the YAML format. It describes the container parameter, for example, the images to be used, the size of the memory, and ports to be opened. /5/

From the Task Definition, tasks can be initiated within a cluster. If the launch type is Fargate, the task has its own boundary and does not have the same kernel, memory, and CPU resources. /5/

Amazon ECS will guarantee at least one or at most ten tasks from only one Task Definition simultaneously. If a task fails or stops, ECS is scheduled to launch new task to replace the failed one. This is the scalability and load balancing features. Therefore, it is very efficient for many cases, for example a sudden large of traffic request to the service. /5/

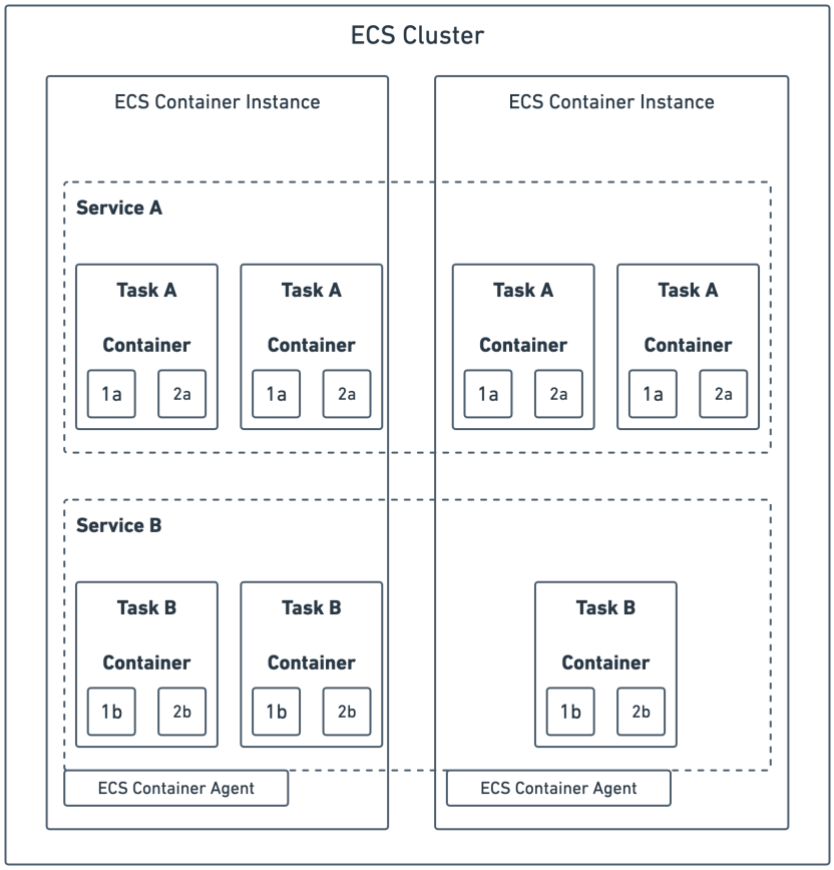

Figure 2. A Cluster has 2 Services, each running a different number of Tasks. /5/

The above figure describes how a ECS Cluster looks like. A Cluster includes multiple Services, Services running at least one to at most ten tasks. /5/

2.1.4 CloudFormation

CloudFormation documentation defines it as: “CloudFormation provides us with an easy way to model collections of related AWS and third-party resources, provision them quickly and consistently, and manage them throughout their lifecycles, by treating infrastructure as code.” /6/

It also says that: “A CloudFormation template describes desired resources and dependencies so that AWS users can launch and configure them together as a stack. Users can use a template to create, update, and delete an entire stack as a single unit, as often as it needs to, instead of managing resources individually”. /6/

This application uses CloudFormation to build AWS stacks such as ECS, ECR, and Secrets Manager and to deploy application containers to ECS. The advantage is that we do not need to go through all AWS stacks and build them manually each time we deploy or update. CloudFormation provides us a way to manage it with a template file, which can be described in YAML or JSON. /6/

2.1.5 Secrets Manager

Secrets Manager is an AWS service that provides a secure store for credential assets such as passwords, keys, and certificates. /7/

So instead of embedding database credentials or API secret keys in the application, we store them in Secrets Manager and retrieve them whenever needed. /7/

AWS Secrets Manager also can be configured to rotate the secret according to a described schedule. /7/

2.2 Quarkus Framework

Red Hat - the company that creates Quarkus defines: “Quarkus is a Java framework designed for Java virtual machines and local compilation. It optimizes Java specifically for containers, making it an effective platform for serverless and cloud

environments”. Quarkus aims to make it easier to build and maintain these applications, and to shorten the time required to get them up and running. /8/

Spring-boot is also one of the most popular Java frameworks, but Quarkus was chosen because it is lightweight and high-performance. It is designed to build and deploy fast and scalable serverless applications without touching too much into configuration.

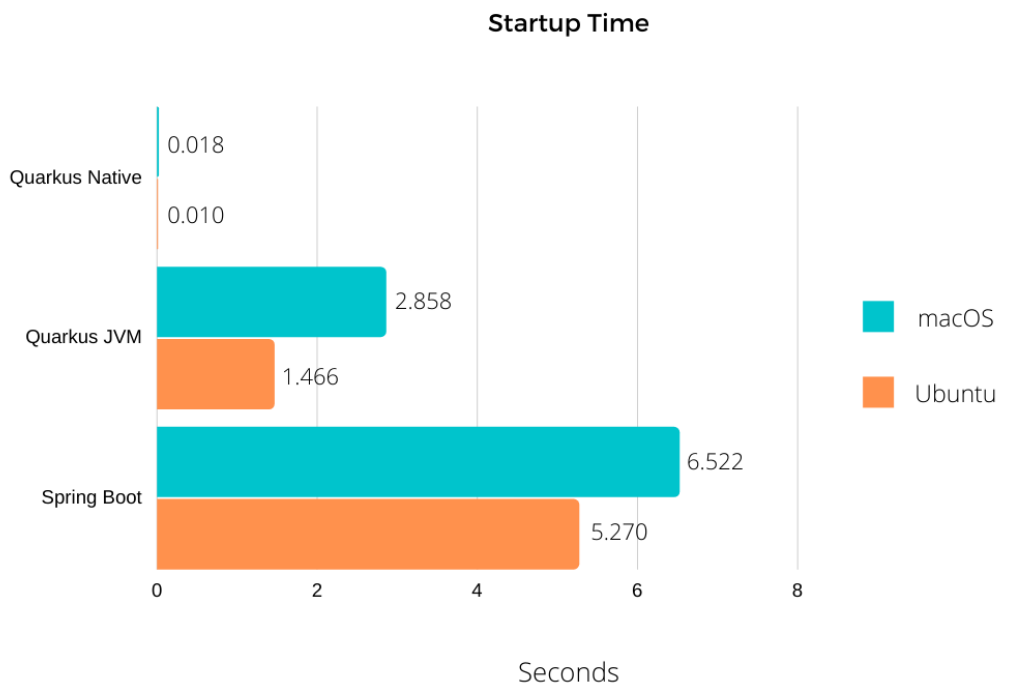

Figure 3. Startup comparison. /9/

Quarkus promises to give a faster startup time more than Spring Boot - a well-known traditional Java framework for building cloud service application. This startup times plays an important role when building a serverless application. /9/

Generally, there are several factors that affect AWS (Amazon Web Services) services pricing, one of those is the startup-time. For example, the more time Tasks in ECS (Elastic Container Service) take, the more it charges. Therefore, minimizing the start-up time offers many benefits from reducing operating costs, to avoid cold start problems. /9/

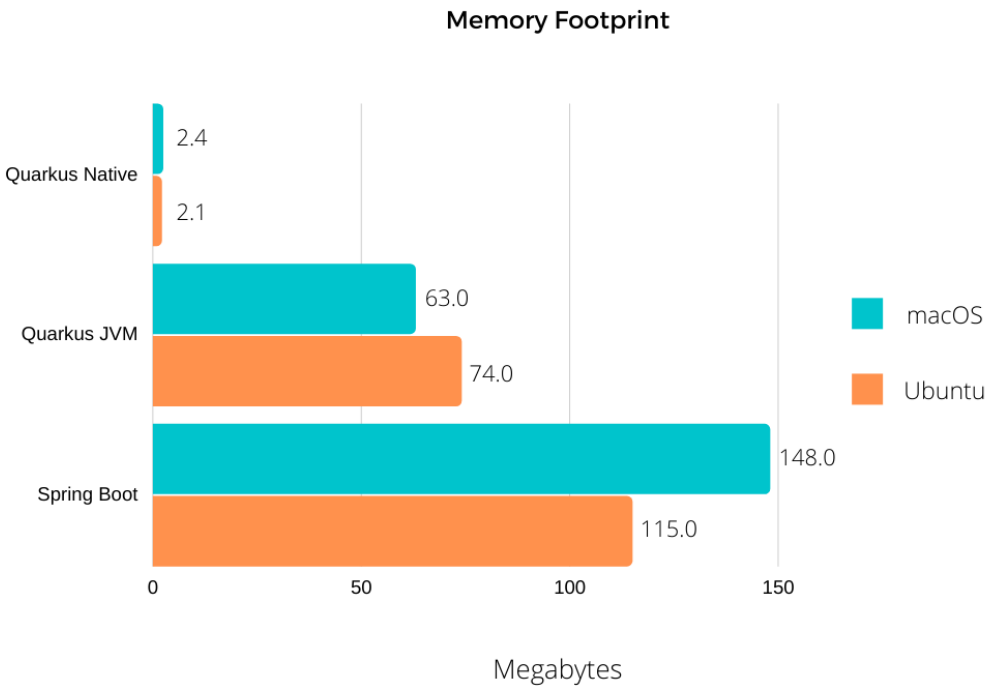

Figure 4. Memory management comparison. /9/

In addition to providing a fast time, Quarkus also provides very low memory consumption. It is also very effective for cost optimization of serverless applications.

2.3 Redis

The Redis cache is an in-memory key-value store - for better performance and reliability. It is a database like any other. What sets it apart from other databases is the ability to store and retrieve data without holding it on a disk. Redis will keep data in memory and provide fast access to it, but if the machine crashes all the data will be lost. To prevent data loss, there is a built-in module that persists data to dump files on the disk. /10/

Unlike Relational Database Management Service (RDMS) such as MySQL or PostgreSQL, Redis has no tables. Redis stores data as key values. It has diverse data types, supports many operations from users. The following is an overview of the data types used by Redis to store values. /10/

– STRING: string, integer, or float. Redis can work with whole strings, parts of strings, as well as increment/decrement values of integers and floats. /10/

– LIST: List is a list of strings, sorted by insertion order. Redis can add an element to two ends of List. The List is suitable for problems that need to manipulate elements near the beginning and the end because this access is extremely fast, even when inserting millions of elements. However, the disadvantage is that accessing the elements in the middle of the list is very slow. /10/

– SET: A set of strings (not sorted). Redis supports operations to add, read, delete each element, check the occurrence of elements in the collection. In addition, Redis also supports set operations, including intersect/union/difference. /10/

– HASH: stores a hash table of key-value pairs, where the keys are arranged randomly, in no order at all. Redis supports add, read, and delete element operations, as well as reading all values. /10/

– SORTED SET (ZSET): is a list, where each element is a mapping between a string (member) and a floating-point number (score), and the list is sorted by this score. The elements of ZSET are sorted from smallest to largest score. /10/

Since the requirement from this application is storing data for a short period of time, Redis is ideal for the application to cache data. /10/

2.4 Docker

Docker is a type of software containerization platform that allows users to isolate and run applications in their own environment. It provides a system for running applications in virtual containers. It is also well compatible with Quarkus and helps build up a much more lightweight service. /11/

In a way, Docker is very similar to a virtual machine. Docker is popular nowadays due to these reasons:

Easy to apply: Docker can be easily used by programmers, and system administrators. It uses containers to build and test quickly. Applications can be packaged on laptops and run on public clouds and private clouds. The mantra is "build once, run anywhere." /11/

Speed: Docker containers are light and fast. Docker containers can be created and start in a few seconds. /11/

Operating environment and scalability: Users can decompose the functions of the application into separate containers. The database example runs on one container, the Redis cache can run on another container, and the Quarkus application runs on another container. Using Docker, users can easily link containers together to form an application, so users can easily extend and update components independently of each other. /11/

To understand Docker, we must know about the Docker image and Docker container.

2.4.1 Docker Image

A Docker image is an immutable file that contains all the necessary information about how the application should be run. This includes server configuration, libraries, and any other dependencies or parameters needed for execution. Docker images are built with a command line tool called "docker build" and contain everything that needed to create a deployable container for the application. /12/

2.4.2 Docker Container

Containers is an instance of an image; it is defined as: “a solution to the problem of how to get software to run reliably when moved from one computing environment to another. This could be from a developer's laptop to a test environment, from a staging environment into production, and perhaps from a physical machine in a data center to a virtual machine in a private or public cloud”. /12/

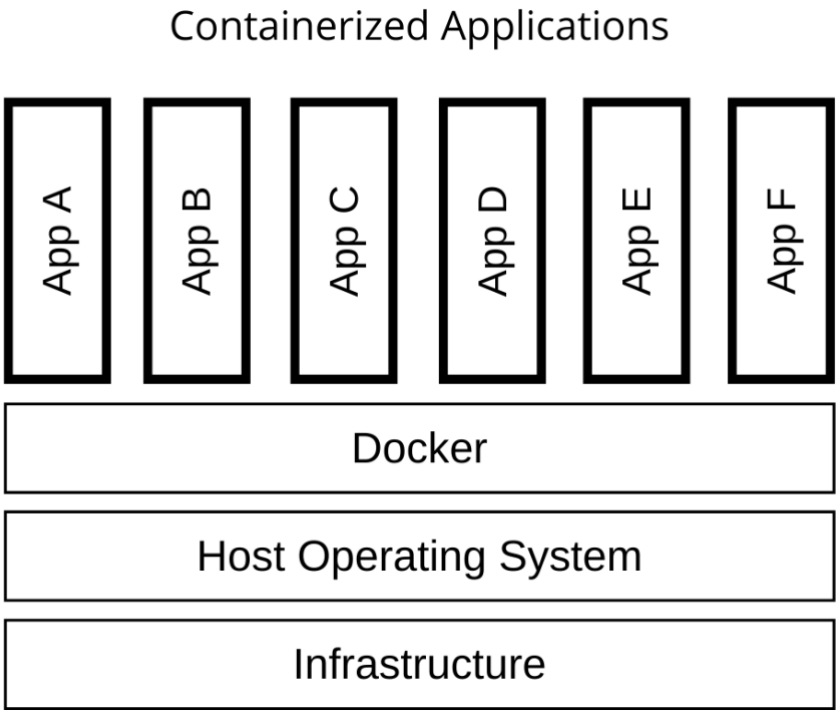

Figure 5. Container application layer. /12/

With the migration to microservices of large systems, Docker is making a hugely important component, making it part of many DevOps technologies.

2.5 Robot Framework

Wikipedia defines that: “Robot Framework is an open-source automation framework for acceptance testing and robotic process automation (RPA). It is an application and platform- independent project with a growing ecosystem of external tools and libraries”. /13/

It was originally shaped by Pekka Klärck – a Finnish developer, then was developed and improved by Nokia. /14/

This thesis demonstrates how a test automation framework could help in developing a software module in general and this application in particular.

3 APPLICATION DESCRIPTION

In this chapter, the structure and requirement specifications of the application will be discussed.

To present a clearer view of this application, this section provides use-case diagrams and class diagrams. This is a visual representation of the structure and use cases of the software.

The following figure below describes the main functions of this application.

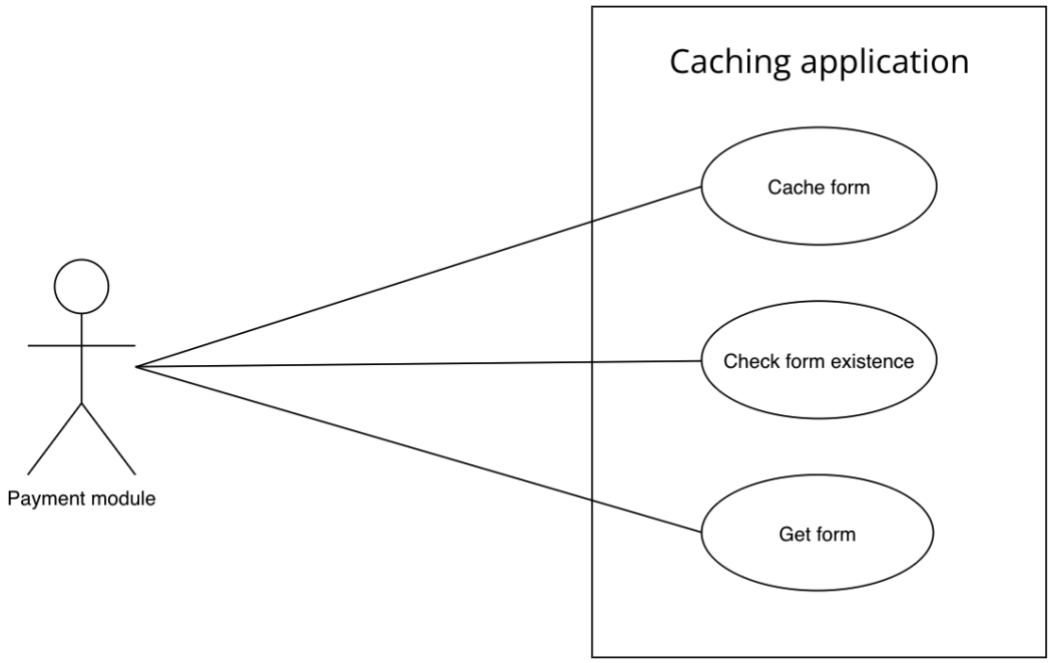

Figure 6. Interaction with payment module.

This caching application is used by another service, which is the payment module. The Payment module uses REST to posting data, as well as to retrieve and check if the data is existed on the cache.

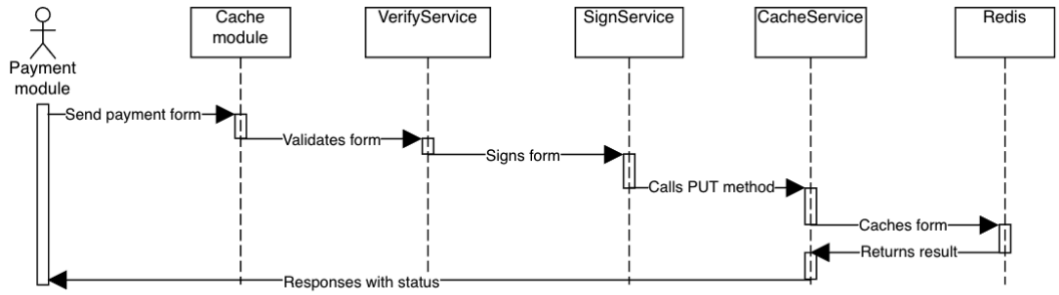

Figure 7. Sequence of interactions when receiving post request.

The above sequence diagram shows various steps from the moment the payment module sends a form to the Cache module to the moment that the form is cached down.

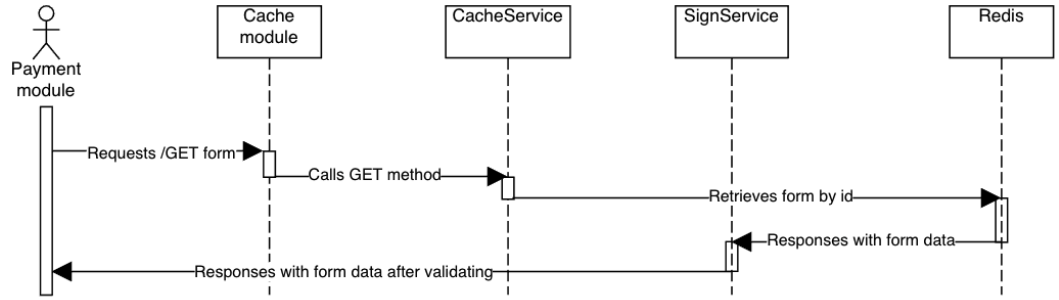

Figure 8. Sequence of interaction when receiving get request.

The above sequence diagram shows various steps from the moment the payment module sends a request to get a form from the Cache module.

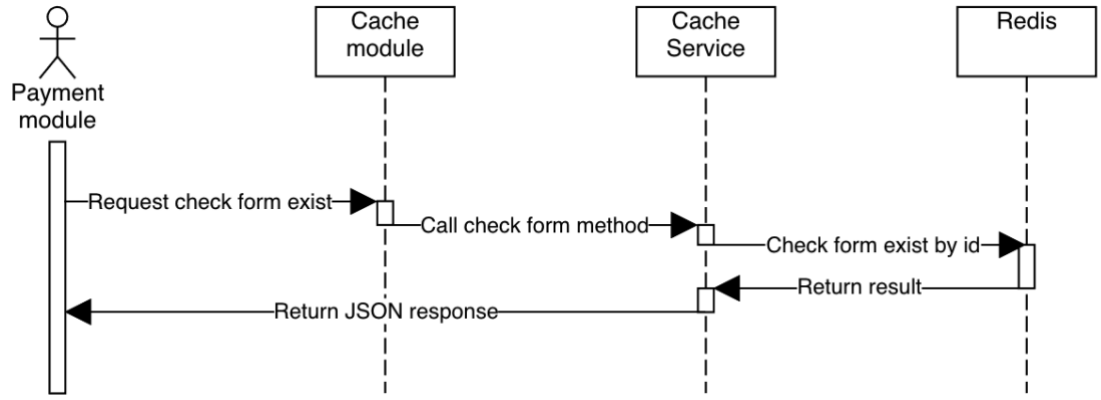

Figure 9. Sequence of interaction when receiving check request.

The above diagram shows steps when Payment module send a checking form request to cache module.

The following object model describes classes, as well as methods and properties which are involved. The following class diagram gives an overview of this software module and the visual relationship between them.

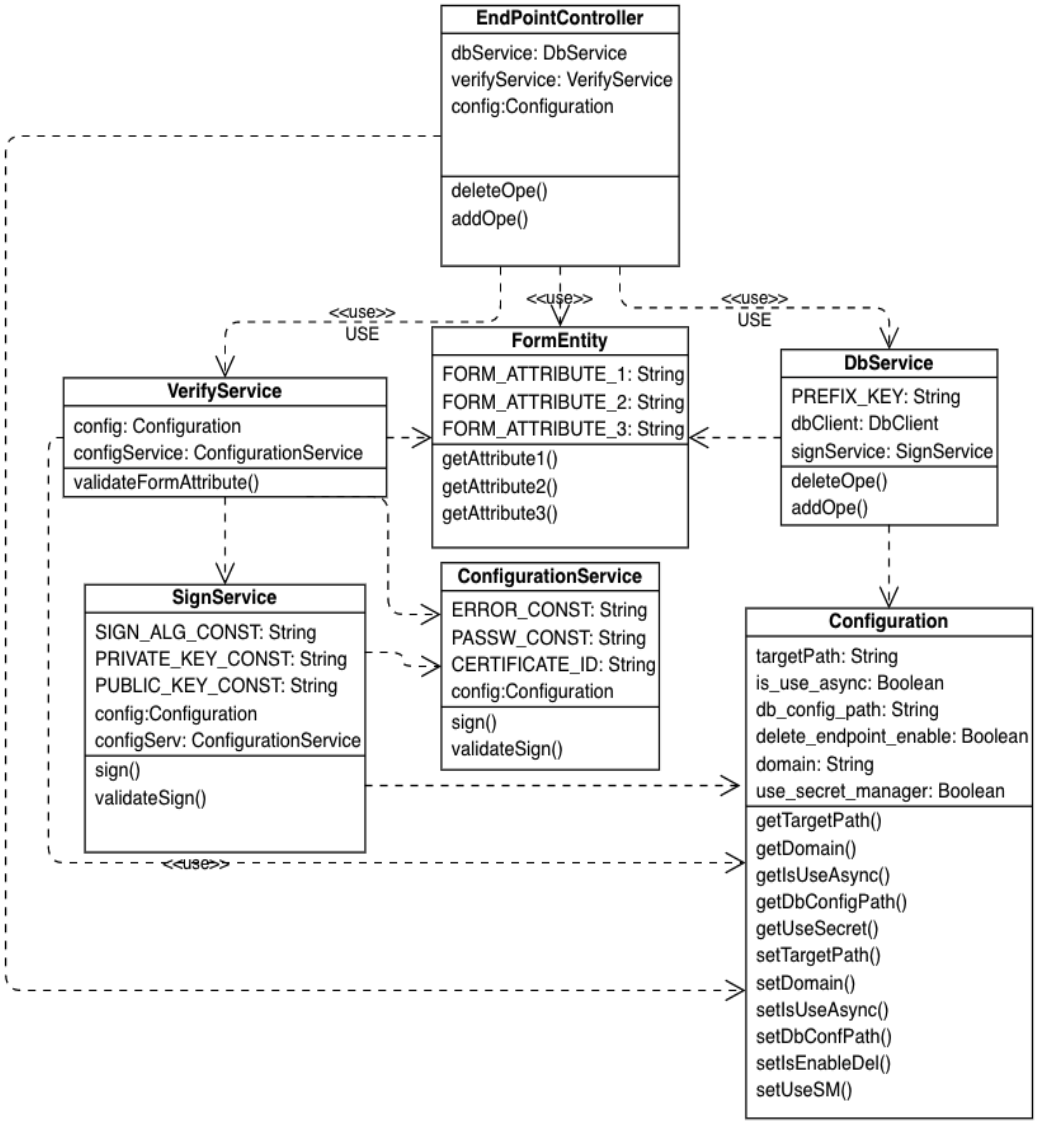

Figure 10. Class-diagram dependency.

The above figure shows a general view of the architecture. In general, Service classes are dependencies of Controller class and Configuration class is a dependency of every class in the system.

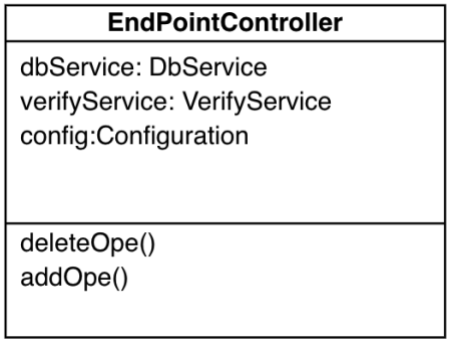

Figure 11. EndpointController class.

The EndpointController class is responsible for receiving a request from the client and performs calling relevant services, for example DbService and VerifyService. It also includes Objects for logging and loading configurations.

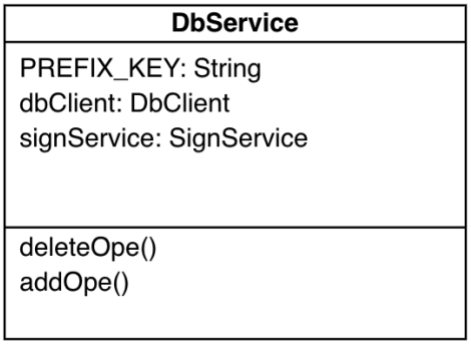

Figure 12. DbService class.

The DbService class is called when the application needs to perform putting data, retrieving data and checking whether data exists on the cache.

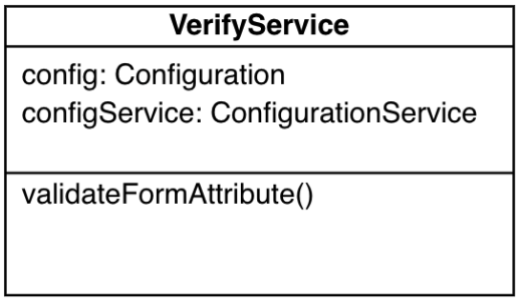

Figure 13. VerifyService class.

VerifyService takes care of the validation task on data, guarantees the received data is in the right format, as well as filters filtering data with malicious objects.

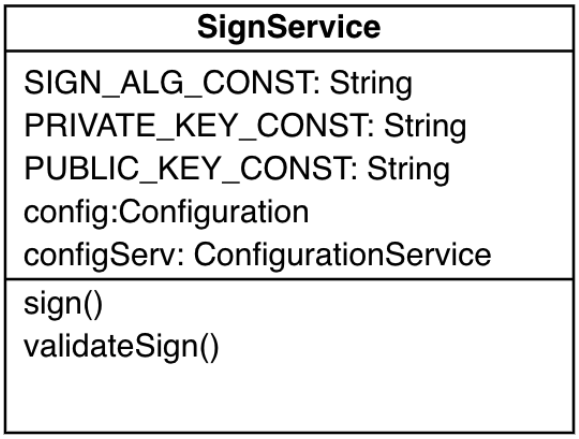

Figure 14. SignService class.

SignService signs the received form data with a digital signature algorithm and validates the signature.

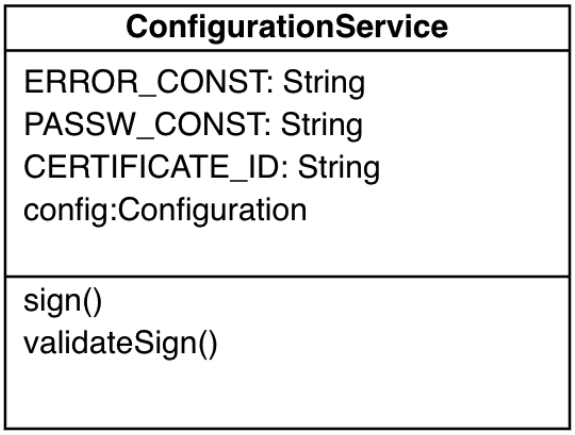

Figure 15. ConfigurationService class.

The ConfigurationService class is responsible for loading configuration from other sources such as Secrets Manager.

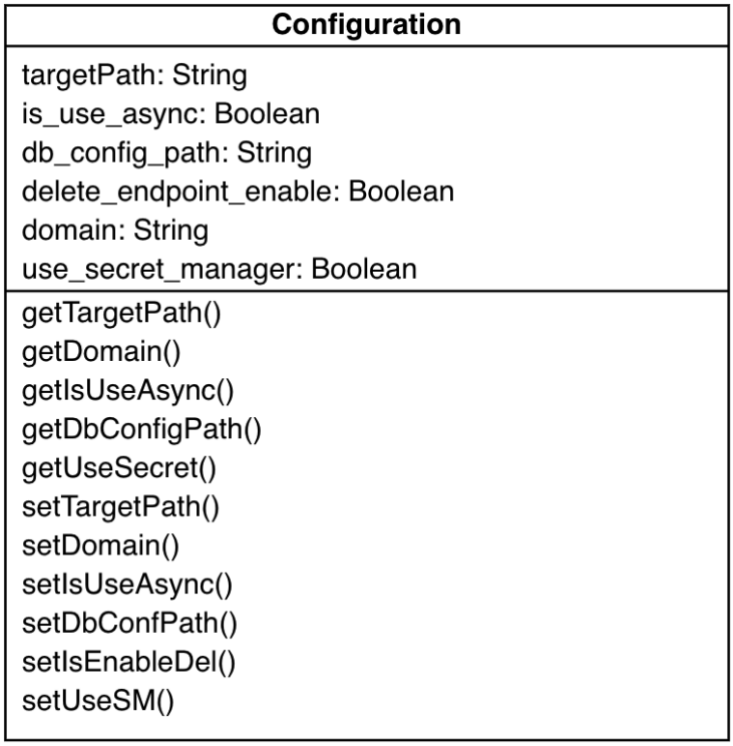

Figure 16. Configuration class.

Configuration values are defined in the application.properties file and can be overridden by an environment variable. The Configuration class provides a standard way to retrieve the values.

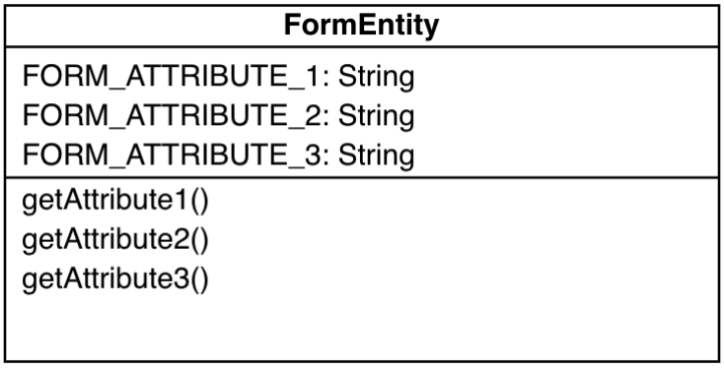

Figure 17. FormEntity class.

FormEntity represents the form sent from the user. It contains all form attributes.

3.1 Requirement Specifications

The requirements can be classified into three levels. Priorities are arranged in the order of must-have, should-have, and nice-to-have. The Must-have requirements are core features of the application, which are essential. The Should-have requirements are important but not necessary for the application. The Nice-to have requirements are desirable and could improve the performance but are not critical.

3.1.1 Must-have Requirements

The Must-have requirements for this application are:

- The time for data to exist in the cache is as short as possible.
- Data must be signed after validation before putting it to the cache.
- Comprehensive automatic tests are mandatory.
- Deploy to AWS as a Docker container.

3.1.2 Should-have Requirements

The Should-have requirements for this application are:

- Application should log events for auditing.
- Data should be encrypted before persisting to the cache.
- Should have a CloudFormation template that could automate the process of building AWS services.
- Data is stored in Redis should be with access control limit.

3.1.3 Nice-to-have Requirements

The Nice-to-have requirements for this application are

- Shell script for workflow automation.
- Docker-compose script for local testing.

4 IMPLEMENTATION

In this section, the process of implementing this application will be described.

4.1 Service Set-up

This application uses Quarkus to build the core service and Docker for containerizing the application. For hosting the container, AWS ECS and ECR are involved. Cloudformation is used for AWS stacks combination.

The general folder structure of the application looks like following figure:

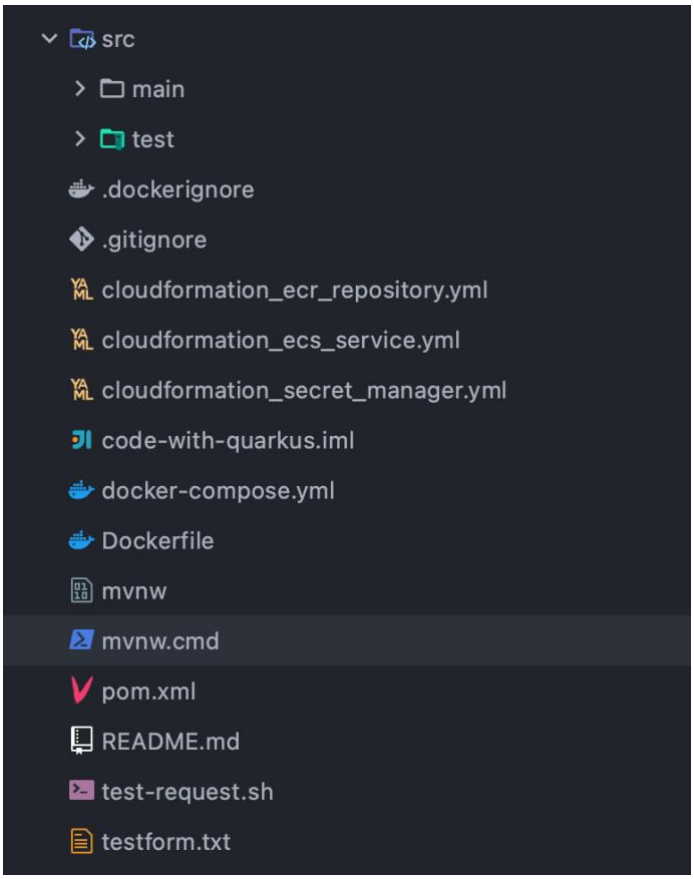

Figure 18. General folder structure

Also, before developing this application, the installation of Java 8, Maven, Docker, AWS Command Line Interface and Python3 was required.

4.1.1 Service Configuration

For configuration properties, putting them in the application.properties file is easy for access throughout the application. Also, we can put them into different profiles properties (application-dev.properties, application-prod.properties, and so on).

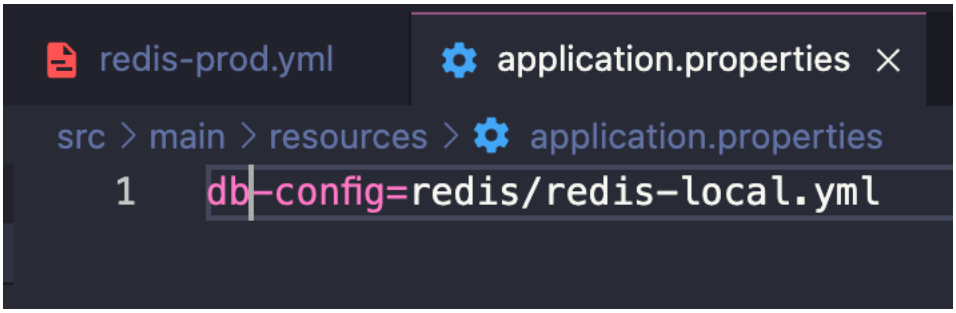

Figure 19. The content of application.properties file.

We define a class to retrieve them through class methods, which annotates with @ConfigProperties.

```
@ConfigProperties
class Configuration () {
@ConfigProperty (name = `db-config`)
    String dbConfig;
    ...
}
```

Code Snippet 1. Configuration class code.

By calling “getDbConfig” method, the method returns dbConfig’s value which is defined in application.properties.

For Redis configuration, before the cacheClient initializes, it loaded a YAML file to config the client. Based on the current environment, it uses different config file.

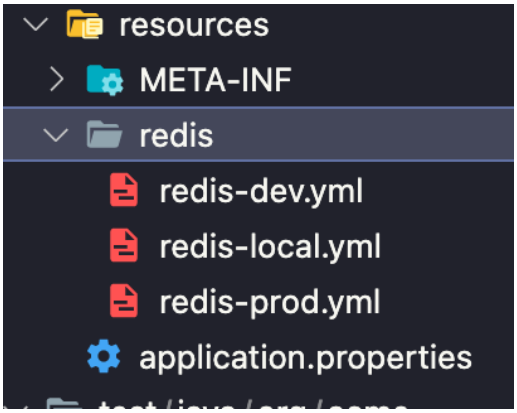

Figure 20. Configuration structure.

Different YAML files are put inside the resource folder.

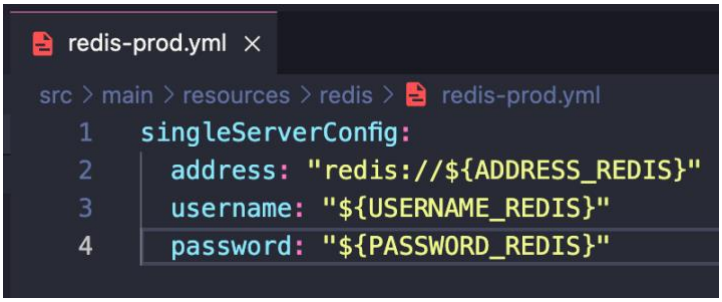

Figure 21. Configuration format.

${ADDRESS_REDIS} can be assigned with environment variable while running.

4.1.2 System Analysis

This application is used to process forms, so at the beginning, we need a form entity class.

```
class FormEntity {
        @[Param]
        @[Constrains]
    String attribute1
    String attribute2
    String attribute3
    String attribute4
    String attribute5
}
```

Code Snippet 2. FormEntity class.

In FormEntity, it contains form properties and adds some Java annotations for constraints (Java annotations are used to trigger functions bound to it without changing the code of the class, method, or property).

If we want the application to be able to receive payment forms from another module or web browser, we will create a class to listen for the requests we want. In this application, we use RESTful for communication.

```
@Path(`api/path`)
class EndpointController {
 verifyService: VerifyService
 dbService: DbService
 logger: Logger
 config: Configuration

 removeOpe (id) {
  IF (check if remove endpoint is not enable)
   throws exception
  ELSE
   dbService removes form_entity associates with id from cache
        return form_entity
 }

 @Post
 addOpe (form_entity) {
  id = [generate random id]
  TRY:
   IF (check if async enable)
    verifyService verifies the form_entity asynchronously
   ELSE
    verifyService verifies the form_entity synchronously
   IF (check if validate finish with OK)
    dbService caches the form_entity
  CATCH:
   return response 500 if exception is born
  return response 200 if no exception is born
 }
}
```

Code Snippet 3. EndpointController.

"@Path(`api/path`)", which an annotation that indicates `api/path` is where the Java class will be hosted. For example, this application is hosted at https://www.exampledomain.io, the web browser can make the request to https://www.exampledomain.io/api/path. Additionally, @Post indicates a

request method should be POST. Now, the application will execute desired task if there is a POST request that come to the application.

The caching service was implemented with the help of Redisson, a Java Redis client library. It provides various APIs to interact with the Redis server, such as opening connections and storing Java objects. The application uses RMap in Redisson to store the client's form, which is based on the Redis Hashes data type. The key of RMap is the id of the form, and the value is the form itself.

```
RMap<String, Form> rMap = redissonClient.getMap(form_id);

rMap.put(form_id, form_entity);
```

Code Snippet 4. An example of how to use Redisson to store Java object to Redis.

4.2 Deployment

This part will show how to build the application into container using Docker. Then it is deployed it onto AWS ECS by CloudFormation template AWS CLI.

4.2.1 Build Configuration

When the Quarkus template is created, it has already included the file Dockerfile.jvm. The multiple parts of the Dockerfile are described as follow.

The header of the file Dockerfile looks as follows:

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:8.4

ARG JAVA_PACKAGE=java-11-openjdk-headless
ARG RUN_JAVA_VERSION=1.3.8
ENV LANG='en_US.UTF-8' LANGUAGE='en_US:en'
```

Code Snippet 5. Header of Dockerfile

FROM keyword pulls the base image for the application's image from remote Docker registry.

ARG keyword gives the possibilities for the user to define a variable that can be passed at build-time.

ENV keyword sets an environment variable, and it will persist when the container is run.

The main part of the Dockerfile is as follows:

```
# Install java and the run-java script
# Also set up permissions for user `1001`
RUN microdnf install curl ca-certificates ${JAVA_PACKAGE} \
   && microdnf update \
   && microdnf clean all \
   && mkdir /deployments \
   && chown 1001 /deployments \
   && chmod "g+rwX" /deployments \
   && chown 1001:root /deployments \
   && curl https://repo1.maven.org/maven2/io/fabric8/run-java-sh/${RUN_JAVA_VERSION}/run-java-sh-${RUN_JAVA_VERSION}-sh.sh -o /deployments/run-java.sh \
   && chown 1001 /deployments/run-java.sh \
   && chmod 540 /deployments/run-java.sh \
   && echo "securerandom.source=file:/dev/urandom" >> /etc/alternatives/jre/conf/security/java.security

# Configure the JAVA_OPTIONS, you can add -XshowSettings:vm to also display the heap size.
ENV JAVA_OPTIONS="-Dquarkus.http.host=0.0.0.0 -Djava.util.logging.manager=org.jboss.logmanager.LogManager"
# We make four distinct layers so if there are application changes the library layers can be re-used
COPY --chown=1001 target/quarkus-app/lib/ /deployments/lib/
COPY --chown=1001 target/quarkus-app/*.jar /deployments/
COPY --chown=1001 target/quarkus-app/app/ /deployments/app/
COPY --chown=1001 target/quarkus-app/quarkus/ /deployments/quarkus/
```

Code Snippet 6. Body of Dockerfile file.

This part is responsible for installing Java and the startup script for the Java container application and giving all permission to that script by **chmod** "g+rwX" command.

Then it performs **COPY** instruction to copy the jar file which is the result from the maven build:

```
mvn clean install
```

Code Snippet 7. Command for maven build

As mentioned before, Maven should be installed in the development machine then it can execute the above command.

Finally, the ending of the Dockerfile should be added:

```
EXPOSE 8080
USER 1001

ENTRYPOINT [ "/deployments/run-java.sh" ]
```

Code Snippet 8. End of Dockerfile

The end of the Dockerfile exposes the port 8080 for network communication purpose. It also set the username to 1001 to use when running the image.

Finally, it sets specific command to run the script when the Docker container is initiated.

There we can use the docker command to create the image and also build the container if we want to see how it looks like in the local:

```
docker build -f ./Dockerfile -t demo:1.0-SNAPSHOT .

docker run --rm -p 8080:8080 -p 5005:5005 -e JAVA_DEBUG=true demo:1.0-SNAPSHOT
```

Code Snippet 9. Command for docker build container.

In local environment, it also needs a mock Redis server running to test, this can be achieved easily by pulling a Redis image from Docker Hub and running that image. This application uses a docker-compose file to run multiple Docker containers:

```
version: '3.7'

services:
  redis:
    image: "redis:alpine"
    command: redis-server --requirepass mypassword
    ports:
      - "6379:6379"
    environment:
      - ALLOW_EMPTY_PASSWORD=yes

  demo:
    depends_on:
      - redis
    build: .
    ports:
      - "8080:8080"
    environment:
      - db_config="redis/redis-dev.yml"
      - PASSWORD_REDIS="mypassword"
      - ADDRESS_REDIS="redis:6379"
```

Code Snippet 10. Docker-compose file for integrating containers.

The file docker-compose.yml is placed in the same folder with Dockerfile and it is executed with the command:

```
docker-compose up
```

Code Snippet 11. Command for docker-compose.

4.2.2 Deploy Configuration

There are two ways to deploy a container to ECS: with the AWS GUI console or by AWS CloudFormation. This application uses the second approach, and the following template is used to demonstrate how to use CloudFormation to provision and build AWS stacks in orderly manner.

Before that, we need to have ECR repositories and secrets which are stored on Secrets Manager. All these AWS services will be created by AWS CloudFormation.

An ECR Repository can be created by the following instructions:

```
AWSTemplateFormatVersion: "2010-09-09"
Description: Demo ECR repository

Resources:
 DemoAppRepository:
  Type: AWS::ECR::Repository
  Properties:
   RepositoryName: demo-application
```

Code Snippet 12. CloudFormation template file for ECR repository.

The template is simply created by specifying the name of the repository to “demo-application”.

Then by using the following command, the ECR can be created without touching the AWS Console:

```
aws cloudformation create-stack --stack-name ecr-demo-repo --template-body
file://./cloudformation_ecr_repository.yml
```

Code Snippet 13. Command for creating ECR from CloudFormation template.

The results should look like this:

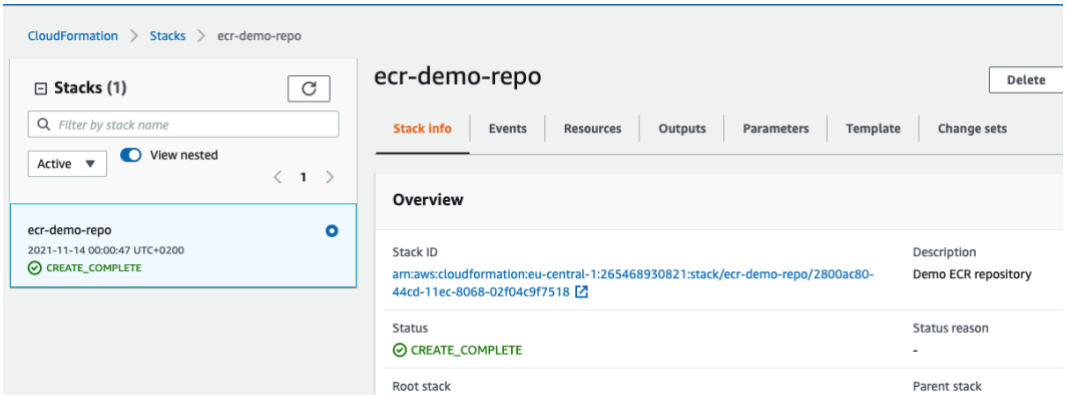

Figure 22. ECR sample result on CloudFormation console.

The above figure indicates that CloudFormation had created a repository successfully. We can easily observe by navigating to the ECR page.

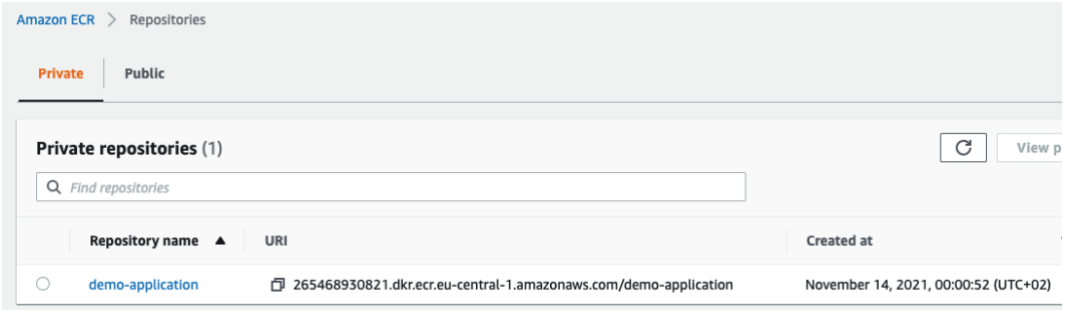

Figure 23. Sample repository on ECR console.

From now on, the image of the application can be pushed onto ECR by the following command:

aws ecr get-login-password --region eu-central-1 | docker login --username AWS --password-stdin [aws_account_id].dkr.ecr.eu-central-1.amazonaws.com

docker tag [your image id] [aws_account_id].dkr.ecr.eu-central-1 .amazonaws.com/demo-application:latest

docker push [aws_account_id].dkr.ecr.eu-central-1. amazonaws.com/demo-application:latest

Code Snippet 14. Commands for pushing local image to remote ECR repository.

The following image shows the image exists in the repository.

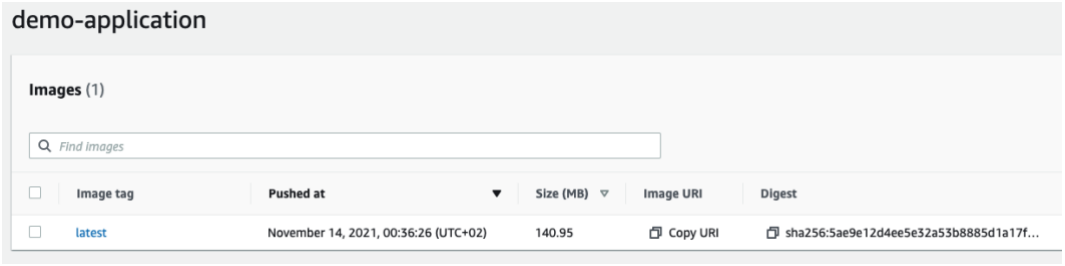

Figure 24. Demo app image exists in the repository after successfully pushing.

The secret can be created on Secrets Manager by the following instructions:

AWSTemplateFormatVersion: "2010-09-09"
Description: Demo Secrets Manager

Resources:
  DemoRedisSecret:
    Type: AWS::SecretsManager::Secret
    Properties:
      Description: Redis password for demo application
      Name: redis-password-demo

```
    GenerateSecretString:
      PasswordLength: 60
      ExcludePunctuation: true

Outputs:
  RedisPasswordArn:
    Description: Redis password
    Value: !Ref DemoRedisSecret
    Export:
      Name: RedisSecret
```

Code Snippet 15. CloudFormation template file for secret creation.

The secret can be created simply by defining the GenerateSecretString property. The length of the password is 60 and it excludes punctuation characters. The Outputs section exposes the value that can be imported into another stack.

The command below creates a secret which is now on Secrets Manager:

```
aws cloudformation create-stack --stack-name demo-secret-manager --template-body
file://./cloudformation_secret_manager.yml
```

Code Snippet 16. Command for creating secret-manager stack.

After executing the above command, a secret with name “redis-password-demo” is put into Secrets Manager.

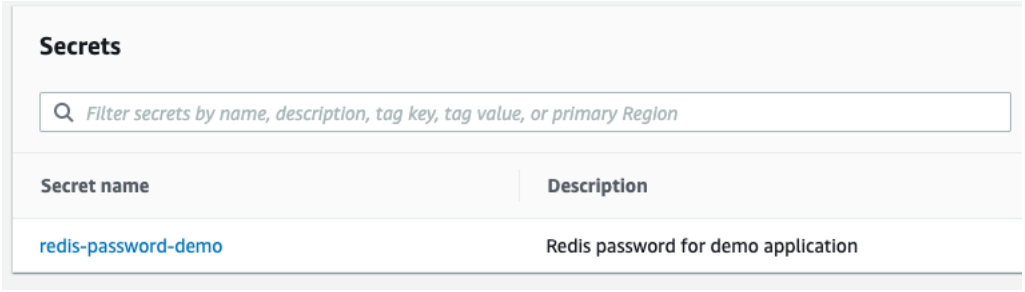

Figure 25. Secret has been created on Secrets Manager.

After the secret and image have been released, the rest of the service can be built. The following template is made to build an ECS cluster, TaskDefinition, ContainerSecurityGroup. The template is written in one file, but this document describes it in multiple parts.

```
AWSTemplateFormatVersion: 2010-09-09
```

```
Description: Demo ecs stack

Parameters:
  DemoApplicationImageName:
    Type: String
    MinLength: 1
    Description: Something like demo-application:1.0.0
    Default: demo-application:latest

Mappings:
  AccountMap:
    "1011011":
      EnvironmentName: dev
      DBConfigPath: redis/redis-dev.yml
      RedisAddress: redis:6379
      SubnetId: subnet-4ca15c30
    "265468930821":
      EnvironmentName: prod
      DBConfigPath: redis/redis-prod.yml
      RedisAddress: redis:6379
      SubnetId: subnet-4ca15c30
```

Code Snippet 17. Heading of ECS template.

Parameters: The template takes the name of the image as a parameter. If no parameters are given, the default value "demo-application:latest" will be applied.

Mappings: Define the values of the different accounts used to deploy the template.

After the **Resources** keyword, all the resources are defined, such as Cluster, ExecutionRole, SecurityGroup, TaskDefinition, ECS service.

```
Resources:
  Cluster:
    Type: AWS::ECS::Cluster
    Properties:
      ClusterName: demo-example-cluster
```

Code Snippet 18. Cluster resource.

Cluster: Create a cluster named “demo-example-cluster”.

The execution role resource for the ECS task:

```
  DemoEcsExecutionRole:
    Type: AWS::IAM::Role
    Properties:
```

```
RoleName: deployment-example-role
AssumeRolePolicyDocument:
 Statement:
  - Effect: Allow
    Principal:
     Service: ecs-tasks.amazonaws.com
    Action: sts:AssumeRole
Policies:
 - PolicyName: AccessSecretRedis
   PolicyDocument:
    Statement:
     - Sid: GetSecrets
       Effect: Allow
       Action:
        - secretsmanager:*
       Resource:
        - !ImportValue RedisSecret
ManagedPolicyArns:
 - arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy
```

Code Snippet 19. Execution role for ECS.

DemoEcsExecutionRole: Defines an IAM role to grant access to Secrets Manager and AmazonECSTaskExecutionRolePolicy permission.

The **Policies** keyword indicates what permission that the Role can have. Here it allows to get a secret from Secrets Manager.

SecurityGroup resource acts as a virtual firewall for instance to control inbound and outbound traffic

```
ContainerSecurityGroup:
  Type: AWS::EC2::SecurityGroup
  Properties:
   GroupName: ContainerSecurityGroup
   GroupDescription: Security group for container
   SecurityGroupIngress:
    - IpProtocol: tcp
      FromPort: 8080
      ToPort: 8080
      CidrIp: 0.0.0.0/0
```

Code Snippet 20. Security Group resource.

ContainerSecurityGroup: Defines the network traffic allowed to access ECS tasks. Here this security group exposes the port 8080.

The **TaskDefinition** resource is used to describe the container that is going to run.

```
TaskDefinition:
  DependsOn: DemoEcsExecutionRole
  Type: AWS::ECS::TaskDefinition
  Properties:
   Family: !Ref 'AWS::StackName'
   Cpu: 1024
   Memory: 2048
   NetworkMode: awsvpc
   RequiresCompatibilities:
    - FARGATE
   ExecutionRoleArn: !GetAtt 'DemoEcsExecutionRole.Arn'
   ContainerDefinitions:
    - Name: demo
      Image: !Sub
'${AWS::AccountId}.dkr.ecr.${AWS::Region}.amazonaws.com/${DemoApplicationImageNa
me}'
      PortMappings:
       - ContainerPort: 8080
      Environment:
       - Name: "db_config"
         Value: !FindInMap [ AccountMap, !Ref 'AWS::AccountId', DBConfigPath]
       - Name: "ADDRESS_REDIS"
         Value: !FindInMap [ AccountMap, !Ref 'AWS::AccountId', RedisAddress]
      Secrets:
       - Name: "PASSWORD_REDIS"
         ValueFrom: !ImportValue "RedisSecret"
       - Name: "ADDRESS_REDIS"
         Value: !FindInMap [ AccountMap, !Ref 'AWS::AccountId', RedisAddress]
      Secrets:
       - Name: "PASSWORD_REDIS"
         ValueFrom: !ImportValue "RedisSecret"
```

Code Snippet 21. TaskDefinition resource.

TaskDefinition: Describe the container and volume definitions of ECS tasks.

- CPU: 1024 (The number of CPU units used by task).
- Memory: 2048 (Task-level memory value).
- Network Mode: awsvpc (Docker networking mode to use for the container).
- Require Compatibilities: Fargate (serverless compute engine).
- Container definition: Template defines what properties the container needs such as port, environment variable, and image to use. Also imports the secret from Secrets Manager, which has been prepared.

This part of the template creates ECS service:

```
EcsService:
 DependsOn:
    - TaskDefinition
    - Cluster
  Type: AWS::ECS::Service
  Properties:
   ServiceName: demo
   Cluster: !Ref Cluster
   TaskDefinition: !Ref 'TaskDefinition'
   DesiredCount: 1
   LaunchType: FARGATE
   NetworkConfiguration:
    AwsvpcConfiguration:
     AssignPublicIp: ENABLED
     Subnets:
      - !FindInMap [ AccountMap, !Ref 'AWS::AccountId', SubnetId]
     SecurityGroups:
      - !GetAtt ContainerSecurityGroup.GroupId
```

Code Snippet 22. ECS resource

EcsService: Defines ECS service that run and maintain tasks.

- Launch type: Fargate
- Network configuration: The network configuration for the service. In this property, we enable a public IP and specify the subnet.

Following command create the stacks that defined in the template:

```
aws cloudformation create-stack --stack-name demo-ecs-service --template-body
file:///./cloudformation_ecs_service.yml --capabilities CAPABILITY_NAMED_IAM
```

Code Snippet 23. Command for creating stacks.

Stacks have been created and the application alive on public IP port 8080. A sample of result is shown:

```
❯ curl http://3.121.186.55:8080/hello
Hello RESTEasy%
```

Figure 26. Result after deploying application.

There, other service can post a form to that domain and the application will perform the caching task.

5 TESTING

Automated testing is a method that uses automated tools to replace human testers to execute test procedures. Then the actual test results are compared with the expected results. In order to save the workload and improve the efficiency of the overall testing software, especially for testing this application, Robot Framework automated testing framework was used. In this section, the process of using Robot Framework is described.

In order to use Robot Framework, the environment must install Python, pip, and Robot Framework libraries.

```
pip3 install robotframework
```

Code Snippet 24. Command for installing Robot Framework

```
pip3 install robotframework-requests
```

Code Snippet 25. Command for installing a robot library

If everything is set up correctly, a simple script can be made.

```
baseUrl = "http://3.121.186.55:8080"

test_form = {
   "PROPERTY_1": "VALUE",
   "PROPERTY_2": "VALUE",
   "PROPERTY_3": "VALUE",
   "PROPERTY_4": "VALUE",
   "PROPERTY_5": "VALUE",
   "PROPERTY_6": "VALUE"
}
```

Code Snippet 26. Variable file.

The code snippet above shows the variables that were needed in the test. The “${baseUrl}” is the service endpoint and “${test_form}” is the mock data that the script uses for posting.

The main test script is written in one file. This document describes the script in two parts, which are the heading and the body.

The heading of the script should have these following keywords:

```
*** Settings ***
Documentation   This is a basic test
Library         RequestsLibrary
Library         Collections
Variables       variable/local.py

Suite Setup     Create Session      demo    ${baseUrl}  disable_warnings=${1}
```

Code Snippet 27. Heading of robot script.

At the begin of the script, it creates a HTTP session to a server that is defined in ${baseUrl} variable by “Create Session” keyword.

The body of the script should look like this:

```
*** Test Cases ***
Health Test
   Log to Console  ${baseUrl}
   ${health}=  GET On Session      demo    /hello  expected_status=OK

Post Test Request
   ${header}=      create dictionary   Content-Type          application/x-www-form-
urlencoded

   ${post_reps}=   Post On Session     demo                /save   data=${test_form}
headers=${header}   expected=status=200

   ${form_key}=    get from dictionary ${post_resp.cookies}    form_key

   ${redis_conn}=  Connect To Redis    ${redis_host}               redis_port=6379
redis_password=${redis_pass}

   ${data_redis}=  Get From Redis      ${redis_conn}         ${form_key}

   Dictionaries Should Be Equal    ${data_redis}   ${test_form}
```

Code Snippet 28. Body of robot test script.

The following description explains what purpose of each line in the scripts:

“${health}= GET On Session demo /hello expected_status=OK” - Performs health check.

“${header}= create dictionary Content-Type application/x-www-form-urlencoded” - Creates a request header before posting html form.

“${post_reps}= Post On Session demo /save data=${test_form} headers=${header} expected=status=200” - Posts ${test_form} to server and expects the status of the response will be 200, otherwise it fails.

“${form_key}= get from dictionary ${post_resp.cookies} form_key” - Looks for the id of the form in the response.

The last three lines are to visit Redis to check whether the form has been stored correctly.

6 CONCLUSION

All in all, the goal of the application was to provide a serverless application to perform the tasks of caching and validating data. It must store data for as short time as possible and provide fast processing speed.

The project achieved fully its goals. By using the latest powerful technologies such as Quarkus, Docker, and Amazon Web Services, the application became lighter and faster than expected. Making future enhancement and maintenance will be easier. Since the process of containerizing applications and deploying them as containers to the AWS cloud is a standard way of implementing serverless architecture, many other applications can easily adapt this method. Cloud services in general and AWS in particular can help maintain the stabilization of the system as well as keep the high availability. By splitting this application to a standalone module, it will be easier to maintain it in the future.

Despite all the advantages provided by the the technologies used in this project, there were some challenges during the development of the application. AWS ECS and Cloudformation have some drawbacks such as error tracking. When deploying the application to ECS, if some errors happen, it will be hard to track where the error comes from, and AWS ECS will give very little information on the causes. Although Cloudformation is a great way to develop cloud stacks, sometimes it fails to update and delete the stacks, which increases latency in the development process.

6.1 Future Work

Another alternative for building a compact service is the Rust programming language instead of Java (Quarkus). Rust is a programming language designed for performance and safety. It is extremely fast and minimizes the amount of resource used. The prototype of the compact service implemented with Rust worked amazingly fast while performing the same amount of task compared to Quarkus. However, the Rust users' community is not so wide and the current developers in

the workplace are not familiar with Rust, therefore maintaining Rust applications would be challenging. /15/

In the future, the application will face more traffic. Therefore, a load balancer will be needed to control the network traffic between a group of back-end servers. AWS Elastic Load Balancing could be the solution when AWS services are used by the application.

REFERENCES

1. Serverless advantages. Accessed 21.11.2021. https://nordicapis.com/server-vs-serverless-benefits-and-downsides/
2. AWS. Accessed 21.11.2021. https://aws.amazon.com/
3. What is ECR? Accessed 21.11.2021. https://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html
4. What is ECS? Accessed 21.11.2021. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html
5. ECS Term. Accessed 21.11.2021. https://www.freecodecamp.org/news/amazon-ecs-terms-and-architecture-807d8c4960fd/
6. AWS CloudFormation. Accessed 21.11.2021. https://aws.amazon.com/cloudformation/features/
7. AWS Secret Manager. Accessed 21.11.2021. https://aws.amazon.com/secrets-manager/
8. What is Quarkus. Accessed 21.11.2021. https://www.redhat.com/en/topics/cloud-native-apps/what-is-quarkus
9. Quarkus vs Spring Boot. Accessed 21.11.2021. https://dzone.com/articles/microservices-quarkus-vs-spring-boot
10. Redis data type. Accessed 21.11.2021. https://redis.io/topics/data-types
11. What is Docker. Accessed 21.11.2021. https://www.docker.com/
12. What are containers? Accessed 21.11.2021. https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.html
13. Robot Framework overview. Accessed 21.11.2021. https://robocorp.com/docs/languages-and-frameworks/robot-framework/overview

14. Robot Framework. Accessed 21.11.2021. https://en.wikipedia.org/wiki/Robot_Framework
15. Rust programming language. Accessed 21.11.2021. https://www.rust-lang.org/