BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

REAL TIME DATA PROCESSING WITH STRIMZI PROJECT

ZPRACOVÁNÍ DAT V REÁLNÉM ČASE S VYUŽITÍM PROJEKTU STRIMZI

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MAROŠ ORSÁK**
AUTOR PRÁCE

SUPERVISOR **Doc. Mgr. ADAM ROGALEWICZ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Bachelor's Thesis Specification

Student: **Orsák Maroš**
Programme: Information Technology
Title: **Real Time Data Processing with Strimzi Project**
Category: Information Systems
Assignment:

1. Study principles of projects Kubernetes, Apache Kafka and Strimzi.
2. Study suitable tools and techniques for design and implementation of an application in container environment (e.g. Quarkus, or Spring).
3. Design an application demonstrating usage of Strimzi system. Pay a special attention on its usage for cluster processing of real-time data and presentation to an user.
4. Design a basic set of system and marathon tests covering real application deployment.
5. Implement application designed in point 3.
6. Implement the test set designed in point 4.
7. Evaluate the solution and propose possible improvements of the created application as well as for the Strimzi system. Discuss the implementation complexity and usefulness of all particular proposed improvements.

Recommended literature:

- Strimzi documentation: https://strimzi.io/
- Kubernetes documentation: https://kubernetes.io/
- Apache Kafka: The Definitive Guide: https://www.confluent.io/resources/kafka-the-definitive-guide/

Requirements for the first semester:

- Items 1 to 3 of the assignement.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Rogalewicz Adam, doc. Mgr., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: May 28, 2020
Approval date: May 14, 2020

Abstract

Container technologies become broadly used in modern times. In prevailing, applications made on the micro-service architecture are rising. This thesis analyzes the design of an application that will process data in real-time. Further, the application will be built using state-of-the-art technologies used by world companies like Netflix, Uber. They are using the systems for real-time data processing such as Apache Kafka, and in recent times they raised it on a higher level by encapsulating this system in the container environment, which guaranteeing effortless scalability. Additionally, using the latest native Kubernetes technologies for processing dozens of data with Quarkus and Strimzi. The problem, which arises, is that these types of real-time data processing systems encapsulated in the containers are especially challenging to test. The main goal of this thesis is a proof-of-concept application based on Strimzi project and also show the designed long term test of the application also known as Marathon, which is the ideal demonstration of user conditions.

Abstrakt

Kontajnerové technológie sa v modernej dobe široko využívajú. Vo väčšine prevládajú aplikácie vytvorené na architektúre mikro služieb. Táto práca analyzuje návrh aplikácie, ktorá bude spracovávať údaje v reálnom čase. Aplikácia bude ďalej budovaná pomocou najmodernejších technológií používaných svetovými spoločnosťami ako Netflix, Uber. Používajú tieto systémy na spracovanie údajov v reálnom čase, ako je Apache Kafka, a v poslednom čase ich zavádzajú na vyššiu úroveň zapuzdrením tohto systému do kontajnerového prostredia, čo zaručuje ľahkú škálovateľnosť. Okrem toho využívaju najnovšie natívne technológie Kubernetes na spracovanie mnoho údajov pomocou programov Quarkus a Strimzi. Problém, ktorý sa objavuje, spočíva v tom, že testovanie týchto typov systémov na spracovanie údajov v reálnom čase uzavretých v kontajneroch je obzvlášť náročné. Hlavným cieľom práce je proof-of-concept aplikácie nad Strimzi testami. Táto práca tiež ukáže navrhnutý dlhodobý test applikácie a systému Strimzi, tiež známy ako Marathon, ktorý je ideálnou ukážkou užívateľských podmienok.

Keywords

Clustering, Real time data processing, Strimzi, Apache Kafka, Quarkus, Java, Kubernetes

Klíčová slova

Spracovanie dát v reálnom čase, Strimzi, Apache-Kafka, Quarkus, Java, Kubernetes, Zhlukovanie

Reference

ORSÁK, Maroš. *Real Time Data Processing with Strimzi Project.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Mgr. Adam Rogalewicz, Ph.D.

Rozšířený abstrakt

V dnešnej dobe sú kontajnerové technológie frekventovaným a zároveň veľmi často skloňujúcim slovným spojením. Aplikácie, postavené na monolitickej architektúre sa stávajú raritou. Silným oponentom sa stáva už dlhoročný súper a to architektúra micro-služieb, ktoré je známa predovšetkým svojou skáľovateľnosťou a jednoduchých rožširovaním. Každý deň sa stretávame s datmi v podobe správ, a jeden s kľúčových atribútov je ich spoľahlivosť vďaka systémom, ktoré sú pripravené čeliť neobvyklým situáciam a tým zaručújú ich správnosť a odolnosť.

Cieľom tejto práce bolo vypracovať aplikáciu pomocou ktorej bude možné procesovať v reálnom čase vzorky dát o obrovskom množstve a to i vďaka kontajnerovej technológie Kubernetes, zaručújúcu efektívnu škáľovateľnosť. Mimo iné toto spracovanie bude prebiehať v jadre technológie, zvanej Apache Kafka. Ak si teraz predstavíme silu technológie Apache Kafka a zároveň k tomu pridáme vlastnosti kontajnerového orchestrálu Kubernetes vznikne nám systém Strimzi. Posledné časti, ktoré zohrali veľkú rolu v aplikácií boli komponenty zaručújúce zátaž na systém a zároveň zobrazovanie dát uživateľovi vo forme jednostranovej stránky pomocou aplikačného programového rozhrania google-maps.d

Hlavným prínosom navrhnutej aplikácie bude hlavne využitie v rámci veľmi požadujúcich záťažových testoch. Jedným z hlavných faktoroch známy je hlavne, že neexistuje žiadny jednoduchý spôsob, ako simulovať existujúcu zátaž s projektom Strimzi. Navrhnutý systém sa použije ako komponenta na testovanie aplikácie Strimzi v reálnych scenároch s obrovským množstvom údajov.

Implementácia sa skladala s viacerých podčastí, kde každá mala unikátnu zodpovedanosť. V procese vývoja si musela každá komponenta prejsť viacerými štadiami ako napríklad: návrhom, implementáciou, spustením komponenty pomocou sbalíku .jar, vytvorením obrazu, pushnutia obrazu na externé registre, vytvorením Deploymentu a na konci otestovanie tejto kompomnety v prostredí Kubernetes. Zároveň, tento proces bol zautomatizovaný a jedná sa o klasický postup vývoja dnešných kontainerových aplikácií.

Navrhnutý externý systém verifikoval Strimzi a bol schopný ho v krátkom čase modifikovať. Boli použité najmodernejšie technológie pre architektúru mikro služieb, ktorými sú Quarkus a React. Quarkus pomocou, ktorého sme boli schopný generovať zaťaženie a vyplnenie týchto údajov pre Strimzi. Navyše, tieto generované data boli aggregované druhou komponentou, ktorá vyvolala zataženie systému v inej časti. Po skončení fázy agregácie Quarkus vystavil tieto data uživateľskému prostrediu. React je iba rozšírenie, aby bolo pre testera viditeľné, že sa údaje menia. Hlavnou prioritou je Quarkus s odosielaním a zhromažďovaním údajov od systému Strimzi.

Navrhnutá, aplikácia bola overená na experimentálnej časti kedy bolo možné vytvoriť marathon test, ktorý po dobu jendého týždňa verifikoval systém, kde periodicky každých 30 minút zbieral štatistiky o jednotlivých častiach sýstémul, ktorý bol pod validáciou. Zároveň, vďaka monitorovacím nástrojom ako Prometheus a Grafana bolo možné level výstupu pozdvihnúť na vyššiu úroveň. Týmto experimentom sa preukázalo, že systém Strimi je spoľahlivý a pripravený ku používaniu na zákaznickej úrovňi.

Real Time Data Processing with Strimzi Project

Declaration

I declare that I have elaborated this Bachelor Thesis by myself under the supervision of Doc. Adam Rogalewicz, further information was provided by Ing. Jakub Stejskal and Bc. David Kornel . I have listed all the literary sources and publications I drew from.

.....
Maroš Orsák
May 18, 2020

Acknowledgements

I would like to thank my supervisors, Ing. Jakub Stejskal and Doc. Adam Rogalewicz, for their time. This work is realized in cooperation with Red Hat Czech, s.r.o.

Contents

Chapter 1

Introduction

In the last few years, one can see a lot of new phenomena. Clustering has become a commonly used word, the era of containers overgrows by creating container orchestral and in concentrates on a design of an application based on Strimzi [14] project. One have to face to face a lot of problem. Data is all around us without noticing them. The thesis resolves many factors, which we are currently facing. First of all, there is no simple way to reproduce real traffic with the Strimzi project. This system will be used as a component for testing our Strimzi application, which is created to process dozens of records in a short space of time. Second of all, we choose a case study about air pollution because we think that our world is not aware of how polluted our countries are. As a case study for one proof-of-concept, we choose an air pollution application.

In recent years, every one of us has certainly seen an increasing percentage of air pollution. This growth is caused by several actions invoked by humankind. One of them is vehicle emissions, which are the primary source of outdoor air pollution. Fuel quality is not sufficient in the last years, and the same applies to most of the vehicles. These two factors highly increase the emission rise[1]. A significant proportion of Europe's population lives in areas, particularly in cities, where air quality standards are exceeded: ozone, nitrogen dioxide, and particulate pollution pose a serious health risk.

Fortunately, UN Environment, the World Health Organization[2] and the Climate and Clean Air Coalition (CCAC)[3], focuses on mobilizing cities to implement policies to protect our health and the planet from the effects of air pollution.

The goal of this thesis is to develop an application, which will show the current user state of air pollution in a particular region. The main attribute of this application is to be scalable. To achieve this, we provide the Kubernetes [1] cluster, real-time data processing, where Strimzi will be used. Easy order to handle essential data, we use REST API using technology Quarkus [24]. Finally, the user will be available to see output data easily with React [21]. All used components together could be used for testing of Strimzi project in the scope of marathon tests. Load testing is a variety of performance testing that enables system performance under real-world conditions. This kind of testing helps determine how the application behaves when virtual users are trying to make some load for some time on the system [18].

The thesis is divided into five chapters with the following structure. Chapter 2 contains a general description of the Kubernetes together with the other technologies like the messag-

[1]The United Nations Environment Programme (UNEP) is the leading global environmental authority

[2]World Health Organization acting as a coordination center in international public health.

[3]The Climate and Clean Air Coalition is a willing partnership of governments

ing platform Apache Kafka and the Strimzi [14], which is an Operator for deploying Kafka into Kubernetes cluster. Chapter 3 discusses the basics of Quarkus technology and reveals its perfection and comparison within the other REST API technology, such as Spring. Additionally, in this chapter, we are discussing the chosen frontend technology, which is React. Chapter 4 set out dataset and architecture, which is supported by UML diagrams, designed API, and Kafka topics. Furthermore, Chapter 4 describing used third party APIs like Google Maps and, lastly, view of the single-page website. The subsequent Chapter 5 showing the implementation of the solution provided in the previous chapter. Chapter 6 presents the testing process of the test application from units to the whole system. Finally, Chapter 7 summarizes and concludes the obtained results.

Chapter 2

Fundamentals of Kubernetes, Kafka and Strimzi

In this chapter, we take a close look at the leading technologies suitable for the distributed real-time project. In particular, Kubernetes which taking care of splitting the load. Apache Kafka for real-time processing data used by Strimzi to handle operations inside the Kubernetes cluster.

Nowadays, applications are based on a cloud environment, or we want to be as we called a cloud-native Kafka is not an exception. *We might ask what is Kafka and how to make it cloud-native ?* The answer to these questions is a system called Strimzi[1].

Containers in general

The idea of the containers was published far in the past. The primary purpose of this technology is to be platform-independent and scalable. Moreover, it is an abstraction of the application layer. It does not create a virtual machine but uses the kernel of the physical computer and creates virtualization for the application and library needed. Also known as lightweight compared to the virtual machines.

2.1 Kubernetes

In recent years, we noticed that many container technologies are being used by big companies, such as Google, Microsoft, Amazon, and more. Kubernetes itself can save an enormous amount of time with the features as e.g. self-healing, secret management, load balancing, and more.

What makes this technology so handy and simple is the way how each Kubernetes component is described in it is configuration file. In particular, each Kubernetes cluster component can be described in simple YAML[2] configuration file, which is pretty easy to read.

2.1.1 History

Everything starts with the physical devices as we called computers. We change three times how we manage applications on the top of the operating system along these lines [7]:

[1] Operator for deploying Kafka into Kubernetes cluster - https://strimzi.io/
[2] https://yaml.org/

- physical
- virtualization
- containerization

Physical

The opening phase of how to deploy applications was simply to execute the program on the physical computer. Nevertheless, this proposal was not as practical as it seemed at first. The main issues were scalability, management of servers, security, and price. Besides, you do not want to share memory between five running applications in an identical environment. On the other hand, you do not want to have five physical servers, which cost much money. All of this led to the formation of the Follow-Up phase.

Virtualization

The next phase has solved problems like scalability, security, and also price. This scenario of the applications can run a single machine without sharing memory, which means it is isolated and encapsulated from the outside of the world. Furthermore, you can run four of these virtual machines on the single physical server CPU, and your only limitation is the server resources. These virtual machines are independent of each other, and therefore security is much higher. The main complication is just that each virtual machines consist of its operating system, which consumer some resources and which can be shared between the application.

Containerization

In the last phase, containerization is considered as a lightweight to virtualization. The difference between these two phases is that virtualization using hypervisor[3] to manage all the virtual machines which has operation systems. The container shares the operating system with the server. Similar to virtualization, they have their filesystem, memory, space, and so on. What makes container technologies so colossal, that they are platform-independent like virtualization technologies, but also without having extra operation system.

2.1.2 Container orchestration

We have containers running in our operation system, but we still do not have something to manage it. The questions like *is my application still running* or *how many containers are running the system*, *which configuration has this container* and so on. That is where Kubernetes takes its role.

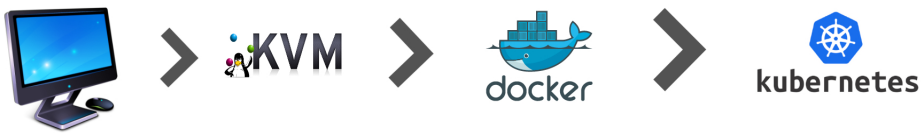


Figure 2.1: Deployment of applications

[3]Hypervisor - It is Software that managing virtuals, for instance, VMware or VirtualBox.

The Figure 2.1 illustrates and summarize phases of deploying applications which started with physical[19], then after a few years the concept of virtualization[23] was revealed. Afterwards, the lightweight era comes with idea whose functionality was based on containerization [20] and finally we have a manager who takes care of the overall management of the individual containers and guarantees their reliability, scales it pretty effectively and more. This is what we call *container orchestration* [1]. It has following features:

- Management of deployments, stateful set, replica set, and custom resource definitions.
- Management of services and load balancing (Service discovery and load balancing).
- Management of storage (Storage orchestration).
- Management of secrets (Secret and configuration management).

Obtaining information of pods, services, storages, secrets and more

With the help of the command-line interface, the management of these features is convenient. The simplest way how to get information about any type inside the Kubernetes cluster is by command *kubectl get resource-name*. This approach applies to all resources. It is essential to note that Kubernetes has a feature called *Self-healing*, which means that if anything inside of the container failed, it would restart it automatically. Additionally, to CLI, it is designed as the REST API, which supports CRUD methods.

Essential Components of Kubernetes

- **kube-apiserver** – validates and configures data for the API objects such as pods, services, replication controllers, and more. REST operations smoothly do everything.
- **kube-scheduler** – making sure that each Pod has matched node so Kubelet can run it. In the next subsection, we take a look at the Kubelet.
- **kube-controller-manager** – as we can see the Kubernetes controller manager is a daemon that encloses the essence control loops shipped with Kubernetes. The controller in Kubernetes is a control loop that follows the shared status of the cluster through the API server and makes changes trying to migrate the current status towards the desired state. For instance, we can imagine these controllers as a replication controller, endpoints controller, namespace controller finally service account controller. [8]

Node Components of Kubernetes

- **kubelet** – this component is used as an observer
  - running containers. (also called *Kubernetes node agent*), runs on all nodes and it has an internal HTTP server allowing read only view on port 10255 with couple of supported endpoints such as */healthz, /pods, /spec*.

- **kube-proxy** – this component is used for the stream forwarding
  - process that runs on all worker nodes,
  - from version 1.2 kube-proxy has permmission to iptables, therefore, the ability to delegate the actual proxy by: relegated to keeping the netfilter rules in syn.
- **container runtime** – this component provides a runtime environment for containers in Pods.
  - Examples are examples: Docker, CRI-O, Containerd, frakt, etc.

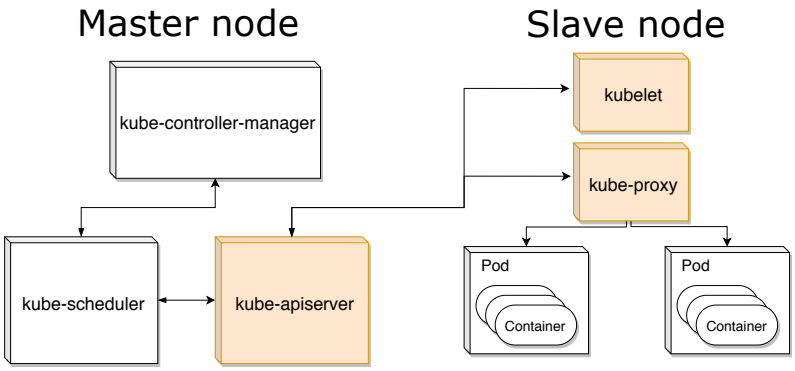


Figure 2.2: Node and essential components inspired by [9]

The Figure 2.2 illustrate architecture, which Kubernetes is using. The master component *kube-apiserver* works like the controller of API calls and communicates with the *kube-scheduler*. It makes sure that every created Pod has some node assigned to run there. Last important component in the master node is *kube-controller-manager*. The main task is to observe and manage all controllers. For instance, *Node controller* controlling the health-check[4], as the name may suggest, also *Replication controller, Endpoint Controller* and more. The worth of mention is that we also have a component called *etcd*, which works as a backup for cluster data. On the other hand, slave or node components as *kubelet* have taken care of containers running inside the Pod. *Kube-proxy*, which reflects all the services defined in the kube-apiserver.

2.1.3 Common objects

Pod

Pod is the smallest unit that can contain application in its container, which can be deployed inside the Kubernetes cluster. Inside a Pod, we can find one or more containers where they share storage, network, and a specification how to run containers. It is considered as a leading resource of the Kubernetes REST API. As an example of the Pod, we can image Kafka running inside the Pod container, which can serve as default Kafka actions like store messages in topics.

[4]Verifying lifetime of Pod over a specified period of time

Service – presents a way how particular components communicate

By default, a service in Kubernetes has a type of *Cluster ip*, which means that communication can be established only inside of the Kubernetes cluster. The way how one is able to expose your application outside of the cluster is to use the following type of service which Kubernetes offers:

- **nodeport** - expose the service to be accessible via node IP with a specific port. For instance, you want to expose your HTTP server to be publicly accessible on a specific port.
- **load balancer** - exposes the service externally using a cloud provider load balance. The load balancer is shown in the definition. *.status.loadBalancer* field, where you can find a real IP address. As an example, if your demands are high and you want an application that requires more ports on specific IPs, then the usage of load balance is a wise choice.
- **ingress** - the previously mentioned types of how to expose a service were service types, but ingress is an entry point for the cluster. *It lets you consolidate your routing rules into a single resource as it can expose multiple services under the same IP address.* [12]

```
apiVersion: v1
kind: Service
metadata:
    labels:
      app: my-new-webserver
    name: my-new-webserver
spec:
    selector:
        app: my-new-webserver
    ports:
    - port: 8080
      protocol: TCP
      targetPort: 8080
    type: NodePort
status: loadBalancer: {}
```

Figure 2.3: Example of service definition using nodeport.

Namespace

This concept of namespaces was introduced in order to run numerous virtual clusters inside one physical. By default, Kubernetes starts with three initial namespaces:

- **default** – the objects which do not have another namespace belongs to the default namespace,

- **kube-system** – namespace for objects created by the Kubernetes system i.e pods, kube-proxy, kube-dns. Furthermore, the service account is used to run the Kubernetes controllers.
- **kube-public** – *this namespace is created automatically and is recognizable by all users (including those not authenticated). In other words, there is a situation we need to have shared resources across the whole cluster; then we have to make sure that theses resources are inside this namespace* [10]

Volume – data storage

The volume is a separated object bind to a Pod. The main ideas behind volumes are following: at first, assume scenario when your Pod crashed and all data will be lost and one would like to retrieve it. Secondly, if one wants to share same data between more pods. The answer to these problems is the *Kubernetes Volume abstraction.*

```
apiVersion: v1
kind: Pod
metadata:
  name: name-of-the-pod
spec:
  containers:
  - image: docker.io/my-new-webserver
    name: my-new-container
    volumeMounts:
    - mountPath: /test-ebs
      name: my-new-volume
  volumes:
  - name: my-new-volume
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

Figure 2.4: Example definition of the Pod, which is using the volume

2.1.4 Controllers

Deployment

Simply said, deployments act as managers, which take care of identical pods. For this reason, they are known as controllers. Without having deployments, it would be tough to manage many pods. As an example, we can imagine the following scenario: The application is released, so with that, we assume that the new image will be available. Only one responsibility would just change the image name inside the deployment file and just confirm that with *kubectl apply -f deployment-definition.yaml.* It triggers the rolling update, and your application is upgraded from one version to another without the user having any clue that something behind the scene is changing. The precondition is that you have at least two replicas of your application running.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  template:
    metadata:
     app: nginx
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Figure 2.5: Example definition of the Deployment

ReplicaSet

This type of controller has to manage the number of pod replicas. Moreover, it ensures that the concrete number of pod replicas is running at whichever time. If we compare ReplicaSet with deployment, the deployment is higher in the hierarchy where it manages ReplicaSet and maintains declarative updates to Pods, which conveys it's better to use deployment instead of ReplicaSet unless you need custom update orchestration on your own. You don't want to update at all. The following Figure 2.6 shows the described hierarchy.

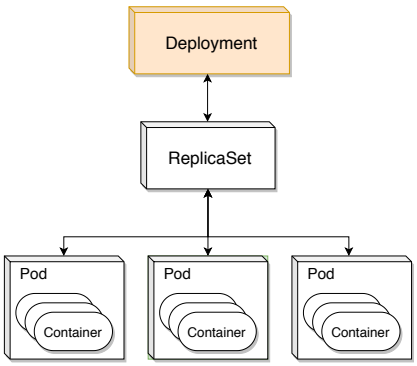


Figure 2.6: Hierarchy of Deployment, ReplicaSet and Pod

StatefulSet

The last notable controller is StatefulSet. The main task of this controller is to provide a unique identity to Pods, for instance, guarantees the order of deployments, scaling which deployment does not offer.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-web-application
spec:
  serviceName: my-web-application-service
  replicas: 3
    spec:
      terminationGracePeriodSeconds: 5
      containers:
      - name: my-web-application
        image: docker.io/nginx:latest
        ports:
        - containerPort: 80
          name: web
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 2Gi
```

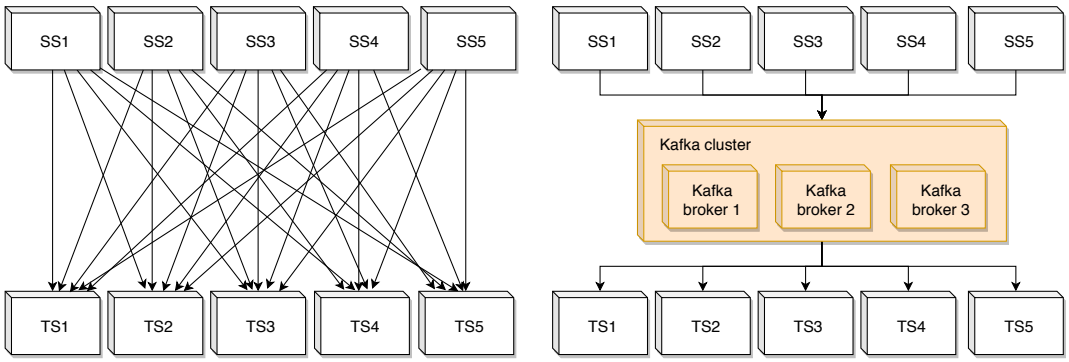
Figure 2.7: Example of StatefulSet definition

2.2 Apache Kafka

Apache Kafka is a streaming platform that offers many features like high performance, distribution, commit log service, and more. It publishes and subscribes to record streams that are similar to a message queue or enterprise messaging system. Moreover, it stores record streams in a robust, fault-tolerant way. Kafka also creates real-time data flows that reliably capture data transferred between systems or applications it can be also used for real-time streaming applications that transform to data streams or response on them. Additionally, widely used by many big companies like LinkedIn, Spotify, Uber, and more.

2.2.1 Motivation behind Kafka

In the past, one had applications or systems that share many data. Moreover, the application was able to provide some useful information to another application. So, there was



(a) Source and Target systems without Kafka (b) Source and Target systems with Kafka

Figure 2.8: Kafka reduce dependencies

one source system and one target system. But what about adding some more source and target systems? Assume an example, where one have five source systems and five target systems. Each source system needs something from each particular target system. Therefore has twenty-five links, which is not effective i.e 2.8 (quadratic complexity). That's why Kafka was born. Let illustrate the same example with ten systems and Kafka is serving as Middleware [5], which placed as the in the middle of these systems. In that case, each source system is bind to the Kafka broker, and all data is delivered by a single link. The following figure 2.8 illustrating described idea.

2.2.2 Terminology

In this section, we take a look at particular Kafka features.

Producer

We can understand producer as application, which publishes messages into Kafka broker. Moreover, his responsibilities are to make sure that he binds to the correct topic on the specific partition. Known, techniques, of publishing are round-robin, where the producer is sending messages to a partition, which is not so active or it is by some self partition method based on the hash table algorithm.

Consumer

The analogy on a producer is a consumer. It is an application, which subscribes to Kafka broker with a specific topic and receives all relevant data from it. The method that consumers use is called polling[6]. In most of cases, data is read from each partition. For more information about the topic and partitions, see the section 2.2.2 and section 2.2.2

Kafka broker

Also called as Kafka server and Kafka node. All of these names refer to the same concept. Kafka broker is a server application, which takes care of all the data that was published into

[5] Middleware – Software, that acts as the middle man between two systems and guarantees interoperability between the systems.

[6] Periodic querying to the server in that case, to the Kafka broker

the Kafka cluster. By contrast to Kafka broker, consumers can fetch data from a specific topics. Only Kafka broker can be scaled to more that one computation unit, which are encapsulated in the so called Kafka cluster.

Kafka cluster

Kafka cluster is a collection of Kafka brokers. Therefore it is fault-tolerant. Brokers share all the data, and if some of them crash, the data will be available because other broker will be running. In the case of all the brokers are down, the data will be lost, but that is so doubtful.

Topic

Kafka topic is equivalent to database table as one can see in figure 2.9. It is not possible to change or to update data if it has already been published. Messages are being stored on a specific topic.

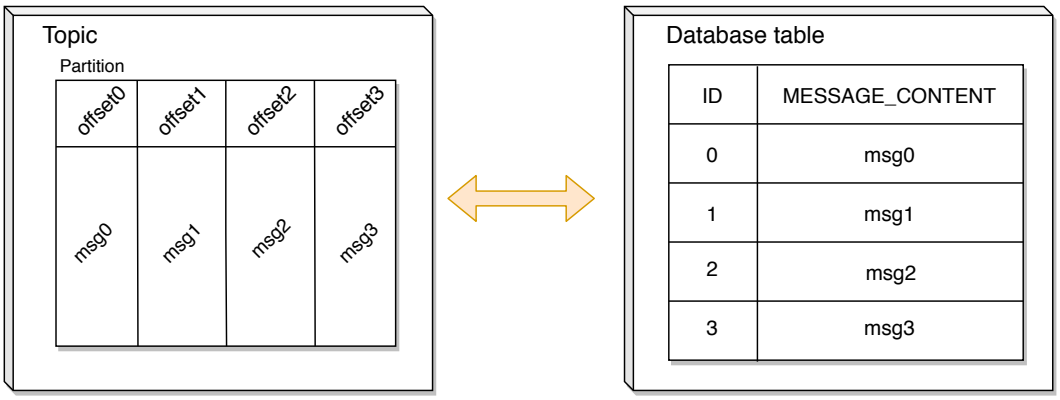


Figure 2.9: Equivalence of Kafka topic and database table

Replication factor

The replication factor is as a number, which defines how many times is the topic replicated to other Kafka brokers. Let us consider the following scenario. We have the Kafka cluster, where we have three Kafka brokers. We create a new topic with an unique name. We have two approaches to how we can create it in Kafka. The first is to create a topic with the command-line application. Secondly, we can create the new topic with applying custom resource definitions 2.3.1. The question can be *what happens if we set higher replication factor then we have available Kafka brokers.* Simply, we are notified that the topic can not be created because we do not have enough accessible Kafka brokers. More about this in 2.3.5.

Partitions

Partitions are as a feature that splits your topic into separate parts. It means that in each partition, we have different data, using this feature we allow an to consumer to fetch data

in a concurrent[7] way. A partition contains offset which serves as an id for the detailed message. In figure 2.10, we can see one topic, which consists of three partitions.

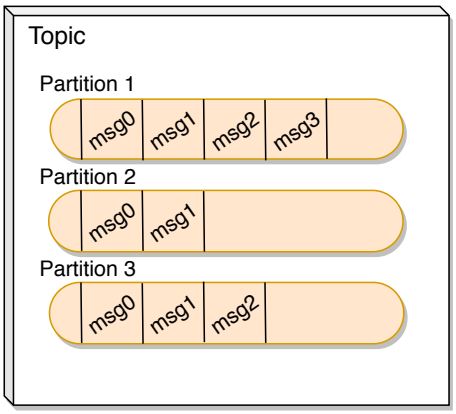


Figure 2.10: Topic with three partitions

Offset

It is an integer value assigned to each consumer indicating the next message, which will be read. Consider the scenario when we have one Kafka broker and one topic with 100 messages. According to offset implementation, it means, that maximum offset value if 100, because it reflects the position of the last message in the topic. If we configure consumers to subscribe to that topic, it uses the polling method and starts with offset e.g. one to zero. The first poll gets twenty messages, so offset move on nineteen and so on. The Figure illustrate this scenario 2.11. In general, we can understand offset as the message index.

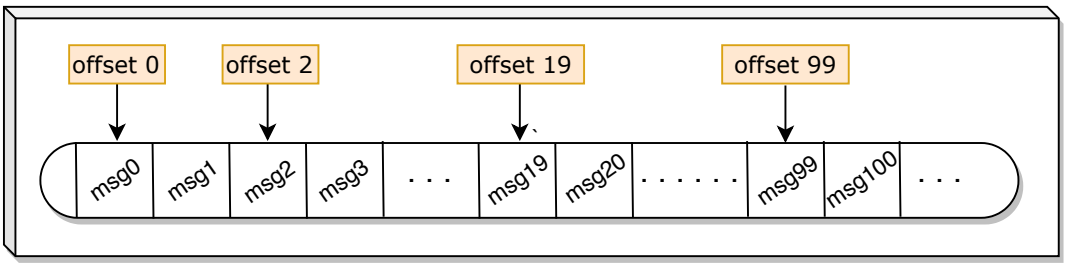


Figure 2.11: Partition offset

We know two types of offsets as follows:

- current - id of the current position of the consumer,
- committed - id of the last successfully saved message

We may also thing another about partition rebalancing. New consumers assigned to the partition should ask a question like *Where to start? What is already processed offset by the previous owner?* Answer on these questions is fairly easy. The consumer use committed offset as the starting point. Its use and starts from the committed offset. Kafka consumer offers two types of how to commit processed message:

[7] Consumes more than one message at the specific period.

- **auto commit** - Enabled by default. Auto commit move offset automatically after a specific period. What is deserving to mentioned is that if we configure our consumer, do a pool request every ten seconds, and our consumer property is set as follows *auto.commit.interval.ms = 11000(ms)*. After the first poll call, we are unable to commit the offset because our auto-commit interval was higher than our poll request, which was only ten seconds. It generally means that if partition rebalancing is triggered, then different consumers must read the same data twice. For instance, when the consumer joins the consumer group, the partition rebalancing is triggered, and the load is reassigned to members of this group. The solution to this problem brings manual commit.

- **manual commit** - As the name suggests, all workloads for management offset is added to the developer's responsibility to take care. This type of commit can be divided into two parts:

  - synchronous approach is straight forward and reliable method, which blocks calls for completing commit operation. Moreover, retry the call if some recovery errors occurred.
  - asynchronous by contrast, we send the request, and it continues executing. The important note is that this type not retry if some failures occurred. *You may ask why we do not retry?* It has some purpose, which we describe in the following sentences. Assume, following scenario. We have one Kafka broker, one configured consumer. Consumers processed one-hundred records, and it wants to commit the offset asynchronously. However, something went wrong and failed for some recovery reasons, and we want to retry after a few seconds. At that time, we are waiting for one-hundred records to confirm. Furthermore, in these few seconds, another call with the consumer is committed with the value of two hundred records, and it is successful while commit one-hundred is still waiting for a retry. In that type of situation, you unquestionably do not want to commit one-hundred but two-hundred, and that is why we are not retrying if something went wrong.

Consumer group

The consumer group behaves as an individual logical unit. Kafka does not support reading from one specific partition with two or more consumers simultaneously. The reason why this concept was created is based on straightforward questions. *How are we able to consumes data concurrently?* Likewise, what is worth mentioning is that we **can not** have more consumers than partitions because, in that type of example, some of them are inactive. This concept differs from other messaging solutions and describing why Kafka is so flexible in comparing with the existing messaging like RabbitMQ.

2.2.3 Publish and subscribe model

Sometimes also called an *Observer*[8] design pattern. General speaking, pub-sub messaging can be divided into these steps:

1. Kafka producers periodically send messages to the Kafka broker into to specific topic.

[8]More detailed about design pattern here https://refactoring.guru/design-patterns/observer

2. Kafka broker stores the messages and assigns to the particular partitions configured for that specific topic,
3. Along with that, Kafka consumers subscribe to a particular topic.
4. Kafka consumer periodically requests for new messages.
5. Kafka forward messages to the consumers,
6. Kafka consumer processes this data and return the acknowledgment of the message, which he processed with the updated offset.
7. this process repeats until we terminate.

Queuing and Publish & Subscribe model

If we imagine the situation where we have all consumers in the same consumer group, the topic messages are load-balanced between consumers. On the other hand, if every single consumer has a different consumer group, messages will be read by every client. The first approach is called the *Queuing model*, while the second one is named *publish and subscribe model*. We can experiment with these two types and design our applications for our needs. The following Figure 2.12[5] illustrating the whole subject of course, neglect the producers, which are producing that data into this topic.

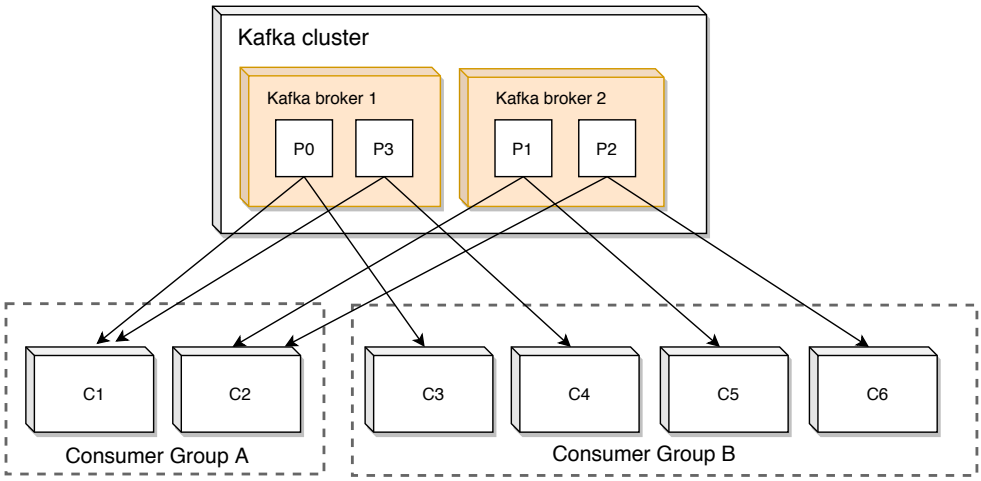


Figure 2.12: Queuing and P & S model

2.2.4 Kafka streams

We can imagine the following problem that you want to filter some data. So far, with current knowledge and understanding of Kafka, you can use Producer & Consumer to filter it. Higher abstraction is Kafka streams, which encapsulating consumers and producers into one unit to reduce time. It is a java application that is highly scalable, fault-tolerant, and so on. Moreover, we do not need to run it on a separate cluster. Usage of this high-level API can be any like Monitoring, Data transformation, and much more. The following figure 2.13 showing whole idea.

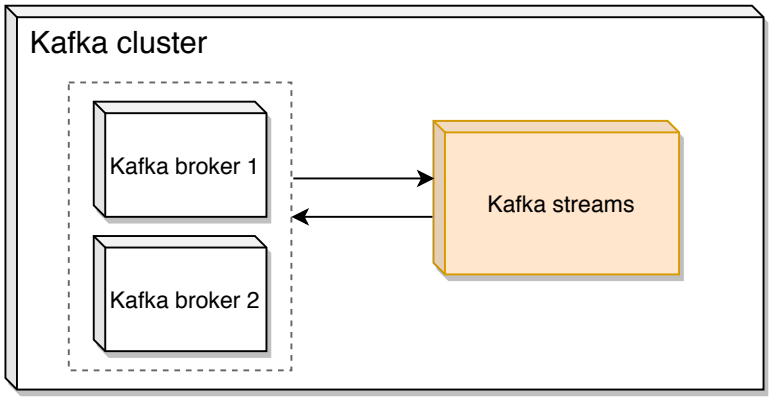


Figure 2.13: Kafka streams

2.2.5 Kafka connect

The essential purpose of Kafka Connect communication with external data sources such as Databases, FileSource, FileSink, ElasticSearch, which is the type of database using CRUD methods instead of the standard SQL language. The connector is an abstraction to a defined unit, which processes data. For instance, the FileSource connector acts as a poll method that if any change happened in the file, it would be automatically pushed to the specified topic in Kafka broker. With the help of the REST interface, it is not so complicated to manage connectors. The offset is managed by Kafka connect, and there is no need to have an additional implementation. Kafka Connect is an abstraction of the consumer and producer API. It has two different implementations, which have many alternatives on how to use it. These APIs are very handy, and usage is straightforward even when you are a beginner. Questions like *How do we pull my data from the data source to my Kafka or How do we synchronize data with my source file.* Answers on these questions has Kafka connect sink 2.14 and Kafka connect source 2.15.

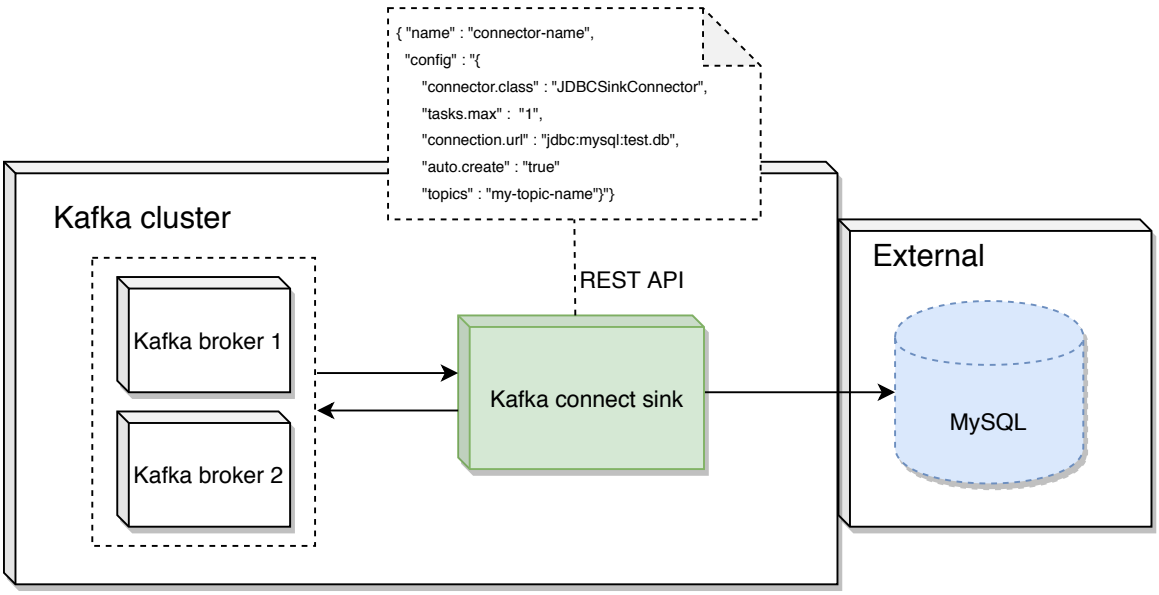


Figure 2.14: Kafka connect sink with database example

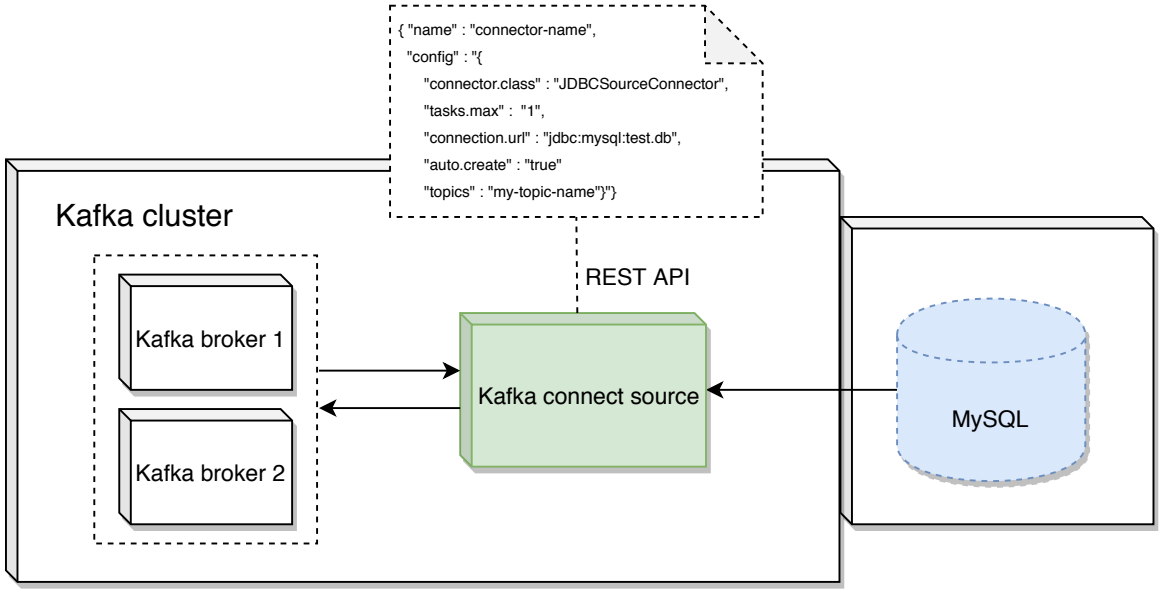


Figure 2.15: Kafka connect source with database example

This figure describes the whole workflow of Kafka connect sink and source. Firstly, if you want to read from the database, you need to use Kafka Connect to sink. I will briefly in points write approach on how to achieve it:

1. setup Kafka brokers,
2. setup Kafka connect,
3. create Sink or Source Connector using Kafka connect REST API,
4. database is automatically created with specified property *auto.create = true*.

Analogically, approach when one needs to write data into a database or any type of storage. Instead of creating a sink connector, you need to create a source.

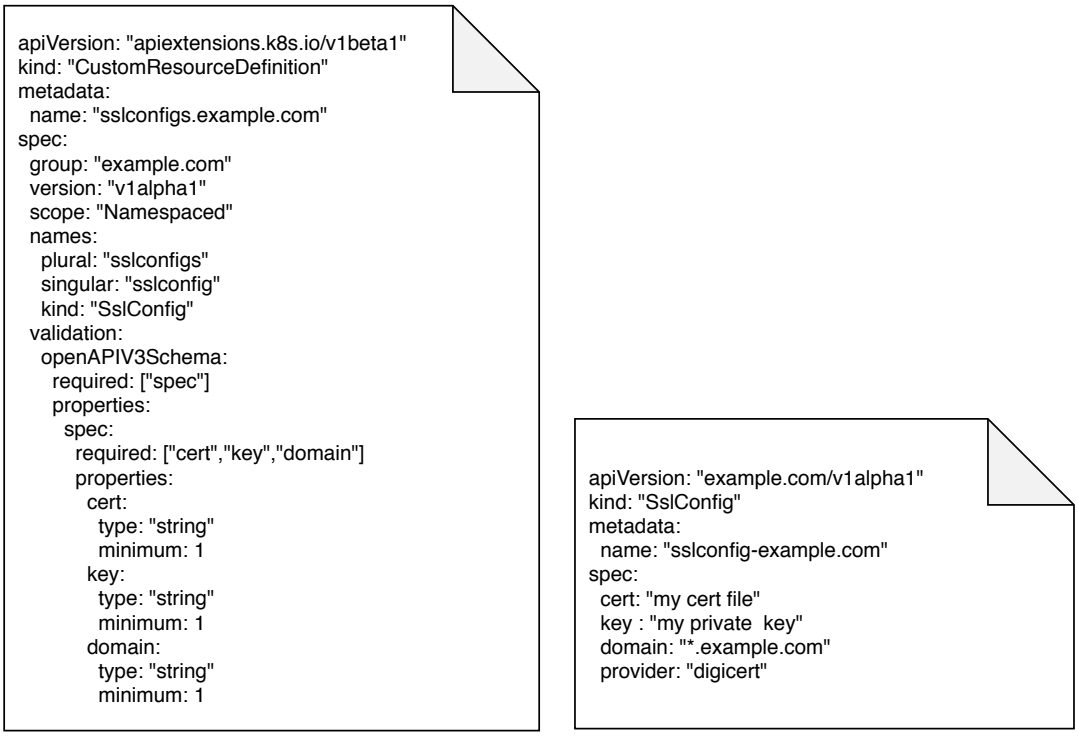
2.3 Strimzi

All of the learned concepts in the previous sections 2.1 and 2.2 were necessarily known for a simple reason. Let us imagine the power of Kafka within a bare-metal server together with the attributes of the Kubernetes container Orchestral. This provides us an ability to build an application called Strimzi.

In the rest of the section, we first introduce the *Kubernetes operator pattern* in common associated with the *Custom resources*. Second we introduce, Strimzi three main operators: Cluster Operator, Topic Operator and User operator (Sections 2.3.4, 2.3.5, 2.3.6).

2.3.1 Custom resources definitions and custom resources

In general, custom resources are the extension of the Kubernetes API. Simply imaginable as many defined objects that serve to our application using CRUD rules. The definition of these objects is captured by stateless YAML files. For instance, we can define our custom resources (called CRDs), (*Custom resource definitions*). As an administrator of the cluster, one can apply these configuration files to enable CRDs in your Kubernetes cluster. In



(a) Example of Custom resource definition [6]

(b) Example of Custom resource [6]

Figure 2.16: CRDs and CR

the following figure 2.3.1, one can see definition of Custom resource definition with type SslConfig and afterward definition of Object SslConfig using CRUD method (*kubectl apply -f file-name-of-crds.yaml*). In this time, one can define your object by CRDs. E.g. a *Deployment* or *Pod* is defined as a *Custom Resource* by Kubernetes developers.

2.3.2 Operator pattern

Another extension to the Kubernetes is a possibility to create your Operators to manage all defined custom resource definitions. Operators provide automation to operations, which your application need to do in a periodical time, such as:

- deploying an application,
- managing backups of the databases,
- selecting a leader for distributed applications without an internal member election process [11].

2.3.3 Architecture

The whole concept of Strimzi is designed by Operators, which we describe in the following subsections. Each Operator has his custom resources definitions used to handle objects created by custom resources. In figure 2.17, one can see that Cluster Operator communicates with Entity Operator and also with the Kafka and Zookeeper clusters. Its fundamental responsibility is to take care of the application. Entity Operator just encapsulates two essential operators, which manages topics and users inside the Kubernetes.

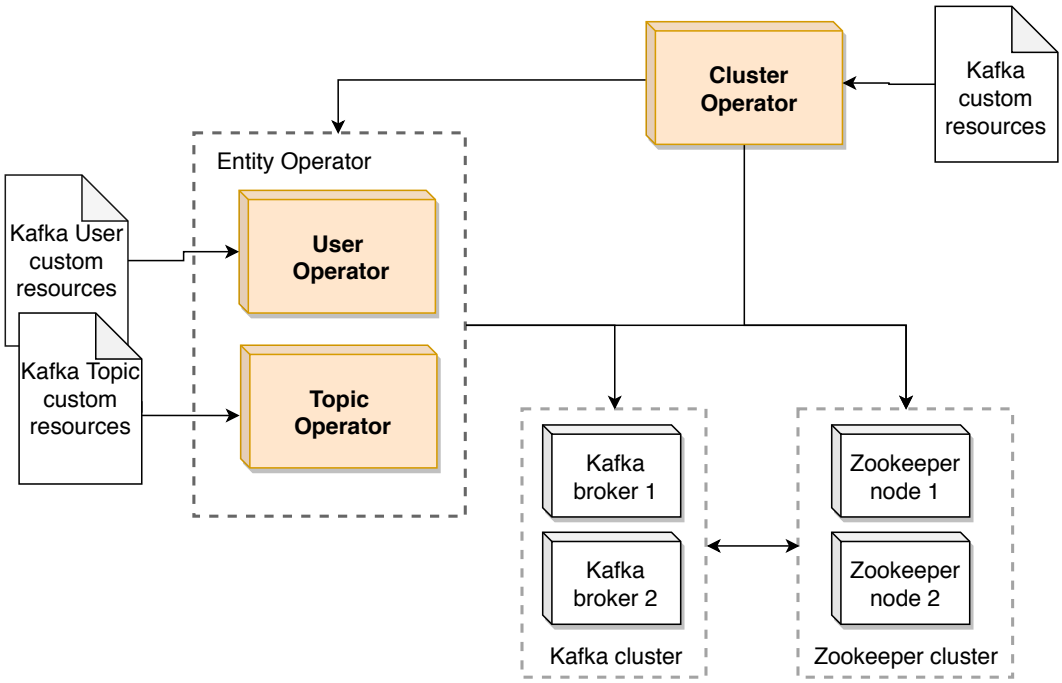


Figure 2.17: Strimzi architecture based on the three operators

2.3.4 Cluster operator

The Hearth of the Strimzi application is the Cluster Operator. Its primary responsibilities are to deploy and manage Kafka clusters within a Kubernetes cluster. Moreover, it can also deploy the topic and User Operator encapsulated in the entity operator[9]. More specifically, it manages and deploys Kafka, Kafka Connect, Kafka bridge [10] and more. It also, manages secrets for encrypted communication using TLS protocol, persistent volume claims, and stateful sets. It is written in Java using Vert.x[11] and Fabric8 Kubernetes client[12].

Architecture

In figure 2.18, we can see Cluster Operator, which takes care of defined Custom resources. In particular, Kafka resources. Additionally, two separate components of Kafka cluster and the Zookeeper cluster are communicating between each other. Firstly, the Cluster Operator creates Kafka custom resource definitions and, after that, applies Kafka custom resources. Furthermore, all of this is wrapped inside the Kubernetes cluster.

2.3.5 Topic Operator

Another essential part of Strimzi is the Topic Operator. It manages creation, deletion, updates, and getting the topic resources. Before these operation can be executed, we have

[9] Operator, which encapsulates the topic and User Operator in one pod.
[10] Proxy server for Strimzi, which provides REST API interface managing Kafka topics, and users.
[11] Open source and event-driven application framework using asynchronous logic more details here https://vertx.io/.
[12] Java client, providing Kubernetes REST API with DSL. In the case of interest https://github.com/fabric8io/kubernetes-client

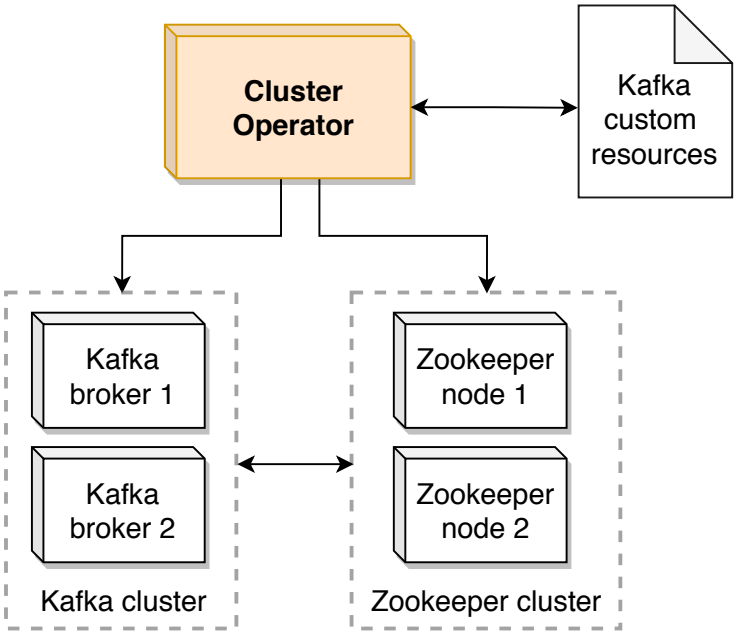


Figure 2.18: Cluster Operator architecture in the Kubernetes cluster

to applied topic resource definitions. Kubernetes resources describing Kafka topics in-sync with corresponding Kafka topics [13]. The three basic scenarios are as follows:

- If an user wants to create Kafka topic then the Topic Operator creates it.
- If the user wants to delete Kafka topic then the Topic Operator deletes it.
- If the user wants to change existing Kafka topic then the Topic Operator updates it.

To create or change topic properties, one can hit some problems of the following. For instance, assume the scenario where the user changes different topic property in Kubernetes but simultaneously in the Kafka itself. Moreover, if the same topic of property is changed at the same time. The first action is considered as allowed, and the solution for this is 3-way diff (more about this method in section 2.19). In general, this method constructs the union of these two differences and find out where the intersection is not empty. The second one is treated as incompatible change. It must deterministically select by some winner policy implemented inside Topic Operator.

The worth of mention is that you cannot change the topic name afterward. In the next sentences, we describe a way of creating a topic using KafkaTopic resources. By default, we can create a topic inside standalone Kafka using command *bin/kafka-topics.sh –create –bootstrap-server localhost:9092 –replication-factor 5 –partitions 5 –topic my-example-topic-name*. Another approach is by means of the defined custom resources. In figure 2.19, we can see the final topic of the custom resource, which has 12 partitions and 1 replica. Let us note that replica cannot be changed afterward. By contrast, changing the partitions is fully supported.

2.3.6 User Operator

The last and significant part of Strimzi is a User Operator. Its principal responsibility is to take care of Kafka users by watching user custom resources and ensure that they

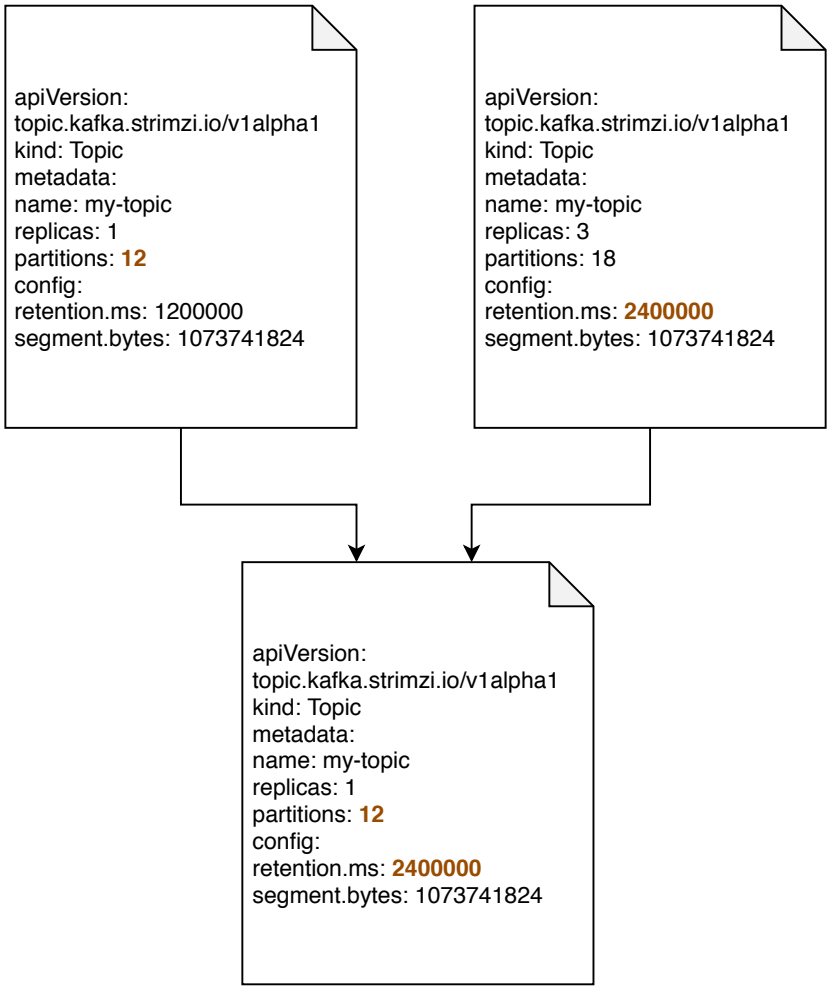


Figure 2.19: 3-way diff method

are configured correctly. Moreover, if the Kafka user is created then the operator creates credentials and store it to the Kubernetes secret. These credentials are then used in the authentication process to consume or produce messages. There are three types how the User Operator can cause trigger some action:

- by the creation of Kafka user, the User Operator creates the user
- by deletion of Kafka user, the User Operator, deletes the user
- by the change of Kafka user, the User Operator, changes the user

Chapter 3

Fundamentals of Quarkus and React

3.1 Quarkus

Quarkus is a framework encapsulating implementation of java. The main goal is to allow, (i) fast boot of application, (ii) native code generation using GraalVM[1]. GraalVM is designed to build native image[2]. Furthermore, it unifies reactive and imperative programming models. Reactive systems, for instance, Vert.x and Apache Kafka. Back in times, the boot-cycle of the application did not matter. It was one physical machine, where we have some applications to serve clients. Redeploying was not so frequent until the era of microservices architecture come. Quarkus was created because with the big bang of cloud-native applications and microservices architecture, where start up time of the application, memory, and CPU consumption matters. All these aspects are essential, and that is why this technology was made to replace old Java. Framework with name Quarkus.

3.1.1 Compile options

The compilation consists of the following phases: (i) scans of keywords, (ii) static and (iii) dynamic analysis. In Java, a compiler named Javac[3] will within the compilation create bytecode for the next usage. At the run time, JVM loads the class files, and determines the semantics of each bytecode. Thern JIT[4] compiler will scan the loaded bytecodes and process some optimization (e.g. reduce redundancy or ,dead variables). It is worth to mention that each option has its own advantages and disadvantages. Different compiler option will be used for long-term running application and diffrent ones for microservices. In the case of microservices, where the application is placed in the container environment, the AOT[5] compilation with Graal VM is technology what one need because of many benefits like faster startup time or speed of response. In the following subsections, we will discuss two alternatives of how Quarkus code can be compiled.

[1] GraalVM is a virtual machine that can be run by many programming languages like Java, Javascript, Python, and so on.

[2] Native image is an executable file that is more efficient and faster to run the application for the first time and simultaneously with a low memory footprint compared to classic jar files.

[3] Javac is a compiler, which includes the best-known toolkit JDK for Java.

[4] JIT, in other words, is a Just in time compiler, which improves the performance of Java programs by compiling bytecode into native machine code in the run time phase.

[5] Ahead of time compilation that translates Java code to native in other words binary code.

Quarkus + OpenJDK Hotspot

Open Java Development Kit [3] is an open-sourced platform project, which implementing Java standard editions. When the code is ready for the customer, it is crucial to create a single executable file. This file, called *Java archive* in short *.jar* is used to aggregate many java classes in one space. The problems that arise with this process are the following: First problem is the file size, which sometimes is very high. Second problem is a slow boot-cycle of the application, which is connected to the response of the first request invoked by the client. Following figure 3.1 illustrating the process of making and running typical application using OpenJDK.

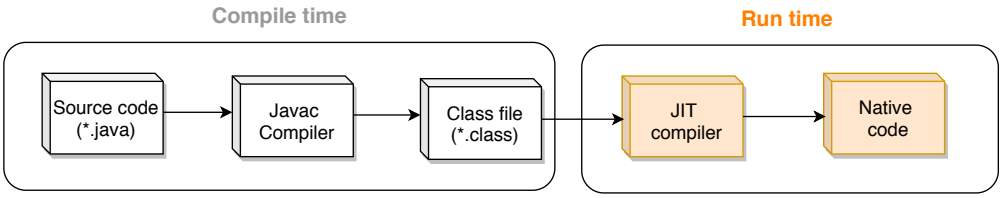


Figure 3.1: Process of making classic java application

Quarkus + Graal VM

An alternative to the Javac compiler is Graal VM [2]. It is designed to compile all types of languages, such as Java, Scala, Python, and C / C++. What makes Graal VM so elegant and different from the classic compilation of java is the option to create a native image. The start of the application, as well as memory utilization, is perfectly optimized with the creation of that image. On the other hand, the time spent on building this image takes much more time than creating a classic jar file. Figure 3.2 illustrates the compilation into the binary code. There is still an occurrence of the JIT compiler used for of the code , which are not executed quite often.

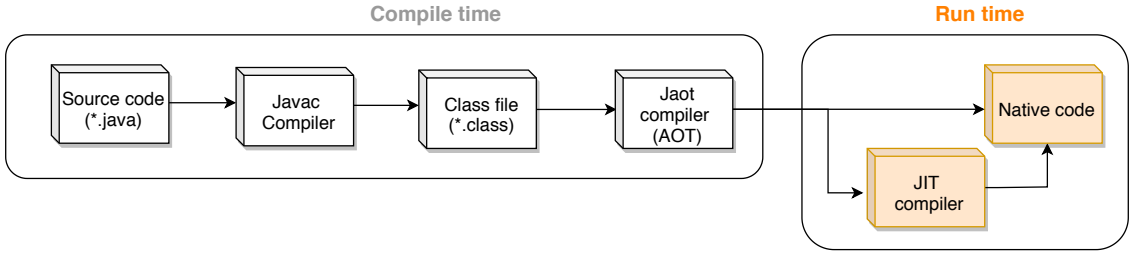


Figure 3.2: Process of making GraalVM java application

JIT vs AOT compilation

In the following figure 3.3, one case see the benefits of Ahead of time and Just in time compilation. The benefits of using a native image, which is based on AOT compilation, is critical for Microservices architecture. Startup speed, low memory footprint and small packaging are mandatory for a container environment, where every other matter. On the other hand, for long term applications, which are not designed as Microservices and require latency and high throughput, then using class JIT compilation is the option.

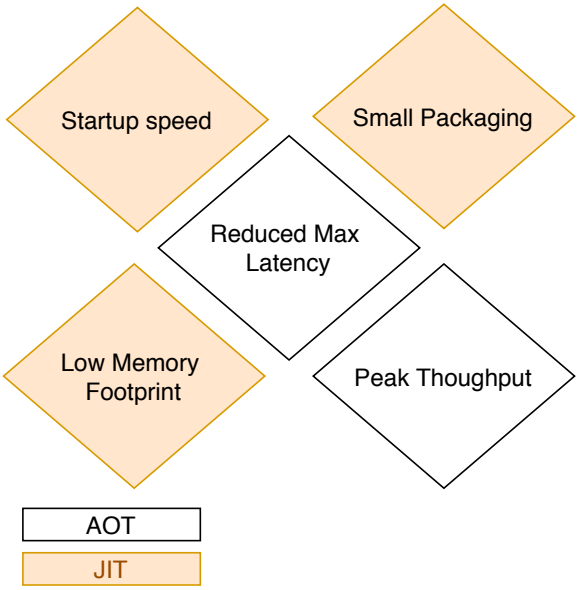


Figure 3.3: Ahead of time and Just in time compilation

3.1.2 Lifecycle

Quarkus is a full-stack framework, designed to run cloud-native applications inside container orchestral. The main idea behind Quarkus is that it starts at the build time instead of run time. Quarkus use context dependency injection as a component model. The procedure for running the application is as follows:

The first phase is compiling all classes using the javac compiler. Quarkus supports Maven and Gradle build systems. If one use Maven, then the application will be compiled by maven-compiler-plugin. Afterward comes the quarkus, respectively quarkus-maven-plugin. Quarkus first analyzes pom.xml [6] to find out on what extensions your application depends as well as loading all configuration files. Once the quarkus has all extensions ready then the „augmentation phase“ will start. Different classes are generated during augmentation. Classes can represents the applications with the help of annotation of @Application-Scoped, which Quarkus provides. CDI bean dependencies are resolved in the Wiring phase. „Assemble“ because it creates a special spring that contains all generated and transformed classes, including classes that define main and many applications. To be more specific, we will describe each of these subsets of wiring and assemble the part. Firstly, everything starts with the load of configuration files in order to get definitions of storages like MySQL, Postgres, Apache Kafka, and more. Moreover, in these files, we can find custom properties for our application. For instance a name of a topic or specific serializer and deserializer class. In the next step, the framework will scan all classes. This means that we will collect all Annotations like @Inject, @Entity, @Incoming, and @Outcoming. Similarly, like Spring [7], Quarkus is considered to be an Annotation Driven Framework. Furthermore, checking getters and setters with related classes are also included. After this process is finished, it will build metadata objects. Like in Hibernate creating with @Entity annotation, we are

[6] pom.xml is a maven definition of the project object model, which specifies all dependencies, versions, names, and other main attributes of the application.

[7] Spring is popular open-sourced Java Framework for developing Java Enterprise Edition applications.

creating table representation of the database. Lastly, the Quarkus prepare reflection and build proxies. In summary, the build process can be split into four points:

1. load configuration files,
2. scan classes as normal annotation driven framework,
3. build metadata objects,
4. prepare reflection.

After this phase, we can choose between JDK or Native mode. Choosing a native image will cost more build time, but on the other hand the application will have, a quick start and low memory consumption. In the case of JDK mode, there are advantages e.g. a better garbage collector, higher peak throughput, Therefore JDK is better option in the case, where your application will be running for a long period. Conclusion of the whole work is illustrated in figure 3.4 and the main steps are briefly summarized as follows:

1. javac compiler (using maven/gradle-compiler-plugin),
2. analyze of dependencies (quarkus-maven-plugin),
3. augmentation & wiring phase,
4. JDK mode or AOT compilation,
5. native mode.

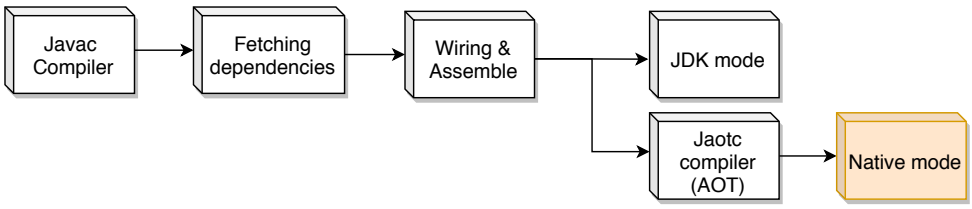


Figure 3.4: Quarkus in steps

3.1.3 Semantics of annotations

Annotations are part of almost every full-stack java framework. In this section, we focus on the semantics of the most important anotations that need to be recognized in the case of this thesis. Many of them are related to Contexts and Dependency Injection, which *allows us to manage the lifecycle of stateful components via domain-specific lifecycle contexts and inject components (services) into client objects in a type-safe way* [4]. Most of the annotations come from Java EE specification and its possibly implementation (JPA, JAX-RS, CDI). Moreover, some annotations are part of the Quarkus API. In Quarkus, we have the only subset of the CDI rules.

@ApplicationScoped

This type of annotation will cause that object defined as @ApplicationScoped is created only once per application cycle. The context is shared between all requests and service invocations. You can also observe the context of the application using onStart and onStop events.

@QuarkusTest & @SubstrateTest

Ideally, these two annotations are nearly analogous. @QuarkusTest is a class annotation, specified on the top of the class as well as @SubstrateTest. The main idea is that @QuarkusTest invoke application on the specific port 9001 before all tests are executed. There is an option that we can randomly generate this port with assigning value 0 to the property. The app will be using classic JDK jar. By contrast, @SubstrateTest is testing the native image of the application.

@Path

@Path applies to class or as method annotation, which has an individual parameter. This parameter maps the path of the endpoint. If we specified @Path(„/something“) to the class and then also @Path(„/more“) to the method, then calling will be delegated from class to @Path(„/something/more“) endpoint.

@PathParam

Parametrized annotation, which creating variability on user input. @PathParm(„isbn“) String id, will search in the database or some type of storage for specific ISBN, which the user will need.

@GET, @POST, @DELETE, @PUT

Represents all CRUD methods, which can be seen in every modern RESP API application. Briefly description, @GET annotation for fetching content. @POST annotation for creating a new object. @DELETE annotation for the deletion of an existing object and finally @PUT annotation for the update.

3.2 React

React was created by the Facebook community mainly by software engineer Jordan Walke. It is written in Javascript language. It is a Javascript library, superset for the Javascript. It's main purpose is developing Single page applications. In the following subsections, we will describe the essential features of React. Traditionally, every web application user interface is created by templates from previously mentioned languages and also HTML directives. React has differentiated from this approach simple by breaking each logical unit, called components. This provides advantages like easy to extend and maintain the pages due to the unification of markup with related view logic. Moreover, javascript can create build abstraction and use basic Object-oriented principles such as abstraction, encapsulation, inheritance, and polymorphism. Additionally, what is worth to mention is that React creates JSX, which stands for Javascript XML. JSX makes React more elegant and readable in both ways. In the following figure 3.5 there is two options how you can define HTML code in your application. My inspiration for this section was mainly from documentation [22] and also w3school web page [15].

```
const reactElement =
React.createElement('p',
  {className: 'farewell'},
  'Have a nice day!'
);
```

(a) Example of React code without using JSX

```
const reactElement = (
  <p className="farewell">
    Have a nice day!
  </p>
);
```

(b) Example of React code with using JSX

Figure 3.5: Javascript XML extension

3.2.1 Virtual Document Object Model

The main feature differentiates Reacts from others, such as Angular, vanilla Javascript, and more template type languages is virtual Domain object model. This means that React creates in-memory own Document Object model kept synchronize with a real web browser. This process is called reconciliation. An important component is the *render()* function. Every change of state of props in reacts component will trigger changes in react tree of elements. His role is mainly reordered virtual DOM to match browser DOM. We know two ways how to implement it. The first approach is to use state of the art algorithms, which has complexity $O(n^3)$, where n is the count of all elements in the react tree of components, which is unacceptable in large applications. The second approach uses a heuristic algorithm based on two rules :

- Two elements of different types will produce different trees. [22]
- The developer can hint which child elements may be stable across different renders with a key prop. [22]

3.2.2 Components

Components can be described as functions that return HTML elements. Its main role is to keep User interface independent and also keep each logical unit in isolation. We know two types of components as follows:

Function components

When one defines the javascript function, be already creates components inside React. The typical behavior of the function is to take props which stands for properties. The function returns Object React element, called function components. Following piece of code illustrating an example of functional components as Javascript function. What is worth to mentioning is that the definition of the element must start with an upper case letter.

```
function FunctionComponent(props) {
  return <p>Common sentense with defined property - {props.name}</p>;
}
```

Class components

Another alternative to define components is by using ECMAScript6 syntax. The advantages of class components compared to function components are as follows. Firstly, the class has its own state and constructor, which is executed in the instantiated phase when the object is creating. Secondly, it creates an abstraction in an object-oriented way, using OOP principles. The following code shows the definition of the class component matching the previous description of the functional element.

```
class ClassComponent extends React.Component {
  render() {
    return <p>Common sentense with defined property - {this.props.name}</p>;
  }
}
```

Both ways of implementation of components has one rendering function. We can show this content to user by calling this function as follows:

```
ReactDOM.render(<FunctionComponent />, document.getElementById('root'));
```

3.2.3 Properties

React Props, which stands for properties in the long term, are arguments of the function and attributes in HTML. Their role is to allow generic and flexible usage of creating customized components. Previous examples illustrated that React has two types of passing properties via functional and class components. Moreover, if we have a class component, where it is explicitly defined constructor, we must make sure that properties are pass to the inherited object via *super(props)* keyword.

3.2.4 State

React state can be seen as properties bound to one particular object created. We know two types of data control in components. The first one consist of properties that are fixed to the parent and the whole lifetime of the object. The second one consist of states, which are used for dynamically changing attributes over a while. When the attribute state is changed, then render function is triggered, and the user can also see the change of HTML code. Example of Clock class component.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {currentTime: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Current time is:</h1>
        <h3>It is {this.state.currentTime.toLocaleTimeString()}.</h3>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

3.2.5 Lifecycle

All Components in React has a feature called lifecycle. The lifecycle can be divided into three phases:

Mount

First part of mounting phase is *contructor()*, which is called when the object is instantiated. What is worth to mention is that we need to pass properties to the parent via reference using the super keyword. Second, is a function called *getDerivedStateFromProps()*. It is a convention, where we set state object based on fundamental properties defined in the contructor. The next part is the rendering phase, where all defined HTML codes is printed to the web site. The final part of mounting is after rendering, in case you need to modify something after the component is rendered. This is defined by function *componentDidMount()*.

Update

The update is triggered when the state of properties are changed. React has the following methods, called in the update phase. The first one is called *getDerivedStateFromProps()*

(same as in the previous phase). This place is to set the state object based on properties. The next one is *shouldComponentUpdate*, where one can create an algorithm, which will decide when your object must update. Default return value of this method is *true*. The third method is called *shouldComponentUpdate*, returns true and triggers rendering part. The React then has to re-render HTML to the Document Object Model. The called *getSnapshotBeforeUpdate()*, allows to you have access to old properties and states, in order to check that the update was correctly done. The last part in the update phases is using method *componentDidUpdate()*, which is executed after the component updates in the DOM.

Unmount

The lowest lifecycle for the component is unmounting. Essentially, this is triggered when the component is removed from the Document Object Model using some deletion operation. Method, which is triggered is called *componentWillUnmount()*.

3.2.6 Events

In every event-driven system, we are dependent on client inputs. React is not an exception, and it is not so provides all event-driven methods as HTML has. For instance click or mouse-hover. All react events are needs to be written in camelCase format and bordered by curly braces like *onClick={makeSomeCallMethod()}* instead of class HTML calling *onclick=„makeSomeCallMethod()“*. We have two approaches on how to invoke events that bind this keyword or without binding. What is worth to mention is that without the binding, this keyword would return *undefined*.

```
// Binding explicitly in the contructor

class Football extends React.Component {
  constructor(props) {
    super(props)
    this.shoot = this.shoot.bind(this)
  }
  shoot() {
    alert(this);
  }
}

// Binding explicitly in the lambda function

class Football extends React.Component {
  shoot = () => {
    alert(this);
  }
}
```

Chapter 4

Design of the application

In this chapter, we cover architecture, dataset, which has been used, along with the design of significant components within Apache Kafka, Kubernetes, and the design of the REST API. Besides, we will describe the frontend of the application that will be accessible to the user. Finally, in the context of Kafka, we will use the Strimzi project inside the Kubernetes cluster to handle operations efficiently using Operators, provided by Strimzi.

4.1 Dataset

The dataset was recommended to this project Simon Woodman, a manager of the engineering team leading the Strimzi project. These, data are known to be the most significant set available in the United Kingdom. They are provided via https://urbanobservatory.ac.uk in various formats such as CSV, JSON for free. This data are free to use. Unfortunately, this set was not enough, and we were forced to create our generator and synthesize more records to simulate a heavy load to the Strimzi system.

The datsa are organized as follows. Each record has a sensor name, name of the gas with related concentration gas value in concrete units, and also a current time when the record was created. Furthermore, longitude and latitude of the sensor position is attached.

4.2 Architecture

The whole concept of this application is to have separated logical units, which will communicate over services. Also known as Microservice architecture. Technologies like Quarkus and Kubernetes are a clear example, where it is mandatory to use this type of architecture. In this section, we will describe how each technology will be involved in the final application. This approach is from the backend side of the application to the frontend and user view,for the designed application.

Build system

Maven is a building system that is among the best alongside Ant and Gradle. Ant applications began early, and after a while, most applications turned to the maven and immediately at the foot of the Gradle. In my opinion, maven is the most stable and straightforward build system that a developer can currently use in terms of configuration and extension. The backend part of the application is using the maven build system. On the other hand, for frontend application is used node package manager.

Application

Kubernetes cluster is holding every single logical unit inside. This cluster will encapsulate the whole application with the integration of Strimzi, Quarkus, and React apps.

Backend

There is several applications within the backend. The **first application**, which will be deployed is **Strimzi**. The main responsibility is to have all data always backed up in case that system went down or any issues like an upgrade of the next version of the application. Kubernetes solves all these needs by increasing the number of pods of the application, which will provide fault-tolerance of the system. Moreover, Strimzi provides high performance, distribution, and commit log service. Additionally, it will simulate the fast transfer of each message, which is provided by smallrye-kafka [1] the connector from the Quarkus. Concisely said, this extension wraps all configuration of producer and consumer console application behavior. Inside the Quarkus application, we need to change application.properties to create a topic with specific commands.

In the following Figure 4.1, one see the whole backend application with three core units.

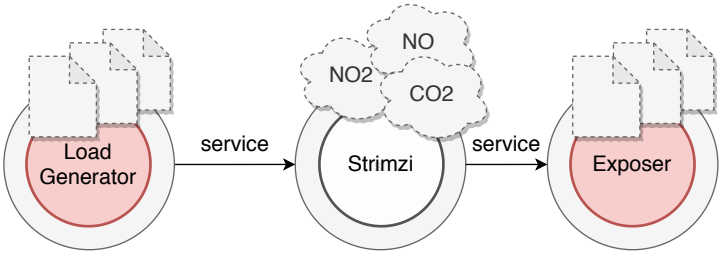


Figure 4.1: Backend architecture of the application

The **next unit** inside the Kubernetes cluster will be already mentioned application based on **Quarkus**. It will periodically query on http://uoweb3.ncl.ac.uk with a specific endpoint. If necessary, it will parse the data set provided by the endpoint, modify then for our purposes and also remove duplicates. Moreover, this customized data will be sent to Kafka and later on retrieve for the aggregation phase. We need only few attributes from the data set, which is the position of sensor expressed by longitude and latitude and the type of gas together with related concentration value.

Aggregated data will be exposed to the particular endpoint and accessed in *http://ip-address:8080/name-of-the-endpoint* by default in 8080 port. These two mentioned technologies make the core application backend.

Frontend

The **subsequent frontend unit** inside Kubernetes is **React**. Based on its role will consist of few parts. First, its main responsibility is that data, which will flow from Quarkus to Strimzi, will be after that process aggregated and exposed by Quarkus on a specific endpoint. Furthermore, React will use the Google API [2] to get maps with related sensors

[1] Smallryle Kafka is a connector with which it is possible to communicate between Quarkus and Kafka with the help of channel streams implemented in Quarkus extension. In the case of interest more can be found here https://smallrye.io/smallrye-reactive-messaging/#_interacting_with_apache_kafka

[2] Google Map API - https://developers.google.com/maps/documentation/javascript/tutorial

and a specific concentration of gasses. The single-page application, also known as SPA, wraps all that we mentioned. The following figure 4.3 shows backend units Quarkus and Strmzi with related communication between Quarkus and React.

In the following figure 4.2, we can see the frontend of the application.

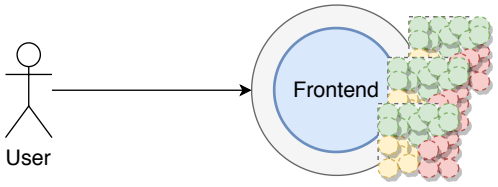


Figure 4.2: Frontend architecture of the application

The **last important component** of the application is the **Google Maps API**. It provides the ability to mark each sensor as one unit with some area. This area will represent air pollution across a few meters. Further, we are able to show to users everything that changes in our data with a form of markers. These markers will have precise info about the concentration of specific gasses done by an uncomplicated info window.

The following figure 4.3 summarizes all the logical units, which I mentioned. Every single circle represents pod, which from the Kubernetes chapter, is the smallest unit, deployable inside the Kubernetes cluster. These pods are communicating through external or internal Kubernetes services.

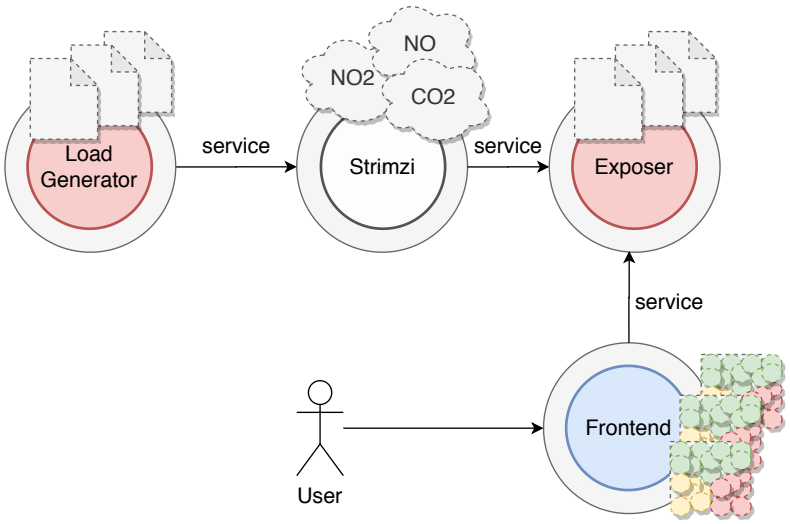


Figure 4.3: Final architecture of the application

Deployment of application

Before we start with deploying an application, there are few things to be done. First, we need images. The image in this context means a unit that encapsulates all dependencies with the build application to be ready in the next phase. After the building phase, we have images ready to be deployed in any container environment such as Kubernetes or Docker.

These local build images are pushed to the external registry *quay.io* [3] The following stage in the deployment of the application is to pull these images from an external registry and apply it to our container environment, which can be done, for instance, by Kubernetes controller Deployment, where inside his definition, we specify the particular image. Prerequisite to be able to pull images from the external registry is to be login by docker client to specified URI. Eventually, when we have all these things ready with related YAML files specifying images, it is just about command *kubectl apply -f deployment-of-application.yaml*. The goal is to have fully automated earlier discussed stages. Done by Makefile [4] file. Everything is shown in the following picture 4.4.

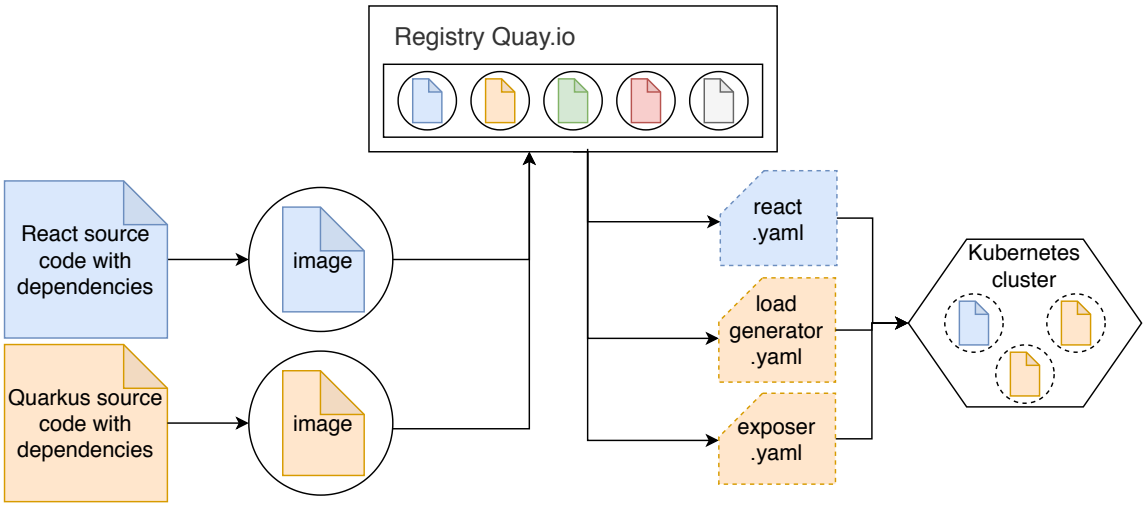


Figure 4.4: Application process

[3]Registry is a server-side application, which can store all kind of Docker images, which leads to a better consistency of using only one space.

[4]Build system Makefile - https://www.gnu.org/software/make/manual/html_node/Introduction.html

Chapter 5

Implementation

This chapter contains a description of each particular component in a more detailed way. First of all, we take a look at Quarkus modules, which are Load Generator, Exposer, and their responsibility to ensure that data is correctly generated, aggregated, and successfully exposed to REST API. Moreover, we take a close look at how Strimzi is contributing to this application. Then, we take a close look at the Frontend of the app, where all data is shown to an user. Lastly, each component will need some configuration to run, which is provided by the Kubernetes cluster. What needs to be mentioned is mainly a change of design during implementation. I found that the data provided would not be very usable within the marathon testing and the load on the system would be almost zero. It was necessary to synthesize the data and create another component called a load-generator.

Prerequisites

Java or GraalVM, if we want native images. Java-script by for those dependencies are to take care of the React npm module. Moreover, the next reliance is Docker, where all images are built and then pushed to the external registry. To be able anyhow to push images, we also need some account, in that case, *quay.io*. Additionally, to adapt to the Kubernetes environment, we need to create Deployment files to our application, where we specify path to the pushed images at the external registry. Finally, we need somehow be able to communicate across the cluster, and even with manual testing, we need to expose to be accessible from outside of the world. Kubernetes service, solves this problem and for this we can use for instance Nodeport.

5.1 Backend components

Load Generator

It is the first module, a component of the application. As previously mentioned, the primary responsibility of Load Generator is to guarantee that receives all data from the Load Generator . We used the data provided by Bsc. Simon Woodman PhD., and with that, we are able to synthesize addtional data in similar format to simulate a heavy load to the Strimzi system. What gives us the advantage to be dependent on external data is the variability to generate data in any way. In general, we can simulate real traffic and use this advantage to create stress tests where the load will sequentially grow, and our only limitation is hardware. In the following Figure 5.1, one can see how the data are structured.

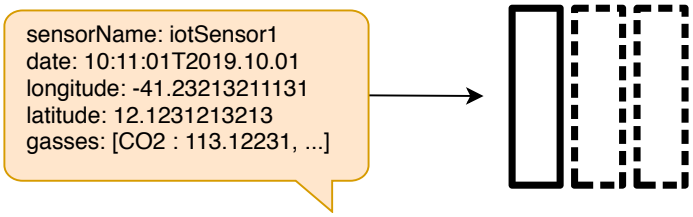


Figure 5.1: Message structure with specific attributes

To be more exact, for the transfer of this kind of data, we used the *channels* provided by the Smallrye connector [1]. The essential to do is the configuration part, which consists of specifying of which connector should be used, to which topic we need to produce data, and lastly, which serializer class we want to use. For clarity, properties can be seen in Figure 5.2.

```
mp.messaging.outgoing.channel-name.connector=smallrye-kafka
mp.messaging.outgoing.channel-name.topic=topic-name
mp.messaging.outgoing.channel-name.value.serializer=serializer-class
```

Figure 5.2: Connector properties of Kafka connect

In Figure 5.3, we can see pseudo code for generation messages in the Load Generator component. The speed of generating of messages depends on GENERATION_INTERVAL. Moreover, in each tick we randomly create instance of message with longitude and latitude regarding on the Newcastle location.

```
@Outgoing(channel-name)
public String generate() {
    Generator.interval(GENERATION_INTERVAL).tick -> {
        recordEntity = new RecordEntity(
                sensorNameRandom,
                randomBetween(longitudeMin, longitudeMax),
                randomBetween(latitudeMin, latitudeMax)
            );

            recordEntity.setGassesArray(gasses);

            return recordEntity.toString();
    }
}
```

Figure 5.3: Pseudo code - Generation of messages in asynchronous way

[1] Kafka connect connector, used for communication with the Apache Kafka, https://smallrye.io/smallrye-reactive-messaging/#_integrating_with_apache_camel

Exposer

The goal of the Exposer is to merge all messages, which is pushed to the Kafka by Load Generator, extract them all, and subsequently aggregated it. Simple as it sounds.

Overall, the flow of the components can be depicted in the following Figure 5.4. Where firstly, Load Generator, through the channel pushing data to the Kafka connect the source and the Kafka, will actualize data in a particular topic whenever the generator pushes new data. On the other hand, Exposer will grab all the data and then aggregated it. Lastly, this aggregated data is exposed by REST API at the particular endpoint with the media-type text/event-stream [2], which behaves like an endless loop, periodically pushing new changes to the specific parameter.

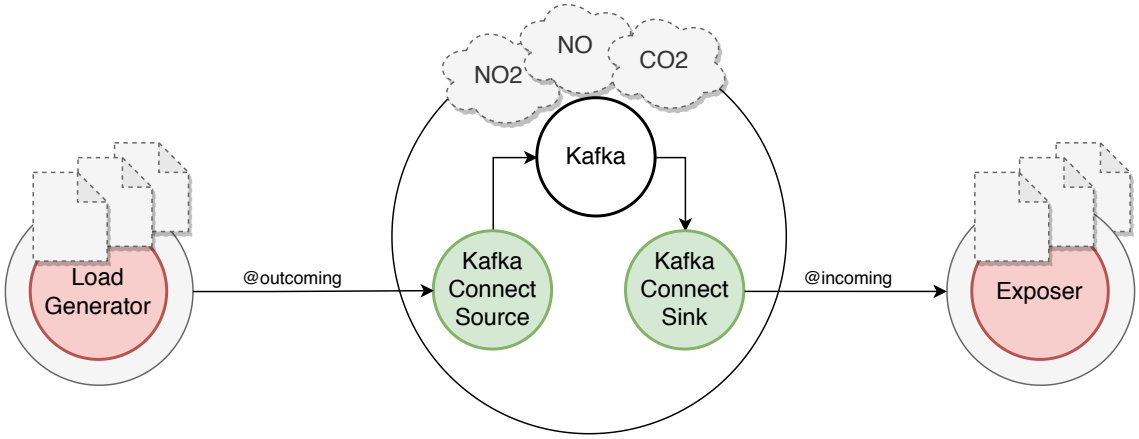


Figure 5.4: Quarkus components in more detail

5.2 Frontend

The frontend side of the application is built by the technology called React. Few interesting things are worth mentioning. Mainly, it is about parsing the data from Exposer 5.1 with the help of the event-source interface, which creates a data tunnel between these points and only ensures that one connection is allocated. Furthermore, what is worth mentioning is the heap-map provided by google-api, which servers and shows a gas concentration of each sensor. Lastly, the location of each sensor is represented by the Marker provided also by google-api.

5.3 Kubernetes deployment

In order to have everything working inside some container orchestral, we need firstly resolve two problems. The first that we need our components encapsulated in the docker images. Secondly, we need these images encapsulate inside the Kubernetes deployments. Once you build the docker images, you can not change it or modify it. Images are defined by a

[2]Event based feature, an alternative to web-sockets more https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events

file called Dockerfile [3], where we specify all the dependencies with additional steps for the instance entry point. In other words, what should be executed after the Deployment of the image is done. When we are done with the building phase, we need to tag each docker image as follows. Conventionally it is *registry-name/organization-name/image-name*. The next step is to make sure that these images are pushed to the external registry, and ultimately, we have defined Kubernetes deployments and services for the communication. In the Figure 5.5, we can see an example of the Deployment of my application. I comment on a few attributes to add clarity to the Figure 5.5:

1. identifying object or controllers inside the Kubernetes domain.
2. metadata for the Deployment of Load-generator, its name to identify the Deployment.
3. replication factor of the pod in this case only one,
4. name of the container running inside the pod,
5. name of the image, which will be executed inside the container.

```
apiVersion: apps/v1
kind: Deployment (1)
metadata:
  name: my-quarkus-load-generator (2)
spec:
  selector:
    matchLabels:
      run: my-quarkus-load-generator
  replicas: 1 (3)
  template:
    metadata:
      labels:
        run: my-quarkus-load-generator
    spec:
      containers:
      - name: my-quarkus-load-generator (4)
        image: quay.io/seequick/bachelor-thesis-quarkus (5)
        command: ["java"]
        args: ["-jar", "load-generator/target/load-generator-0.1.0-runner.jar"]
        ports:
        - containerPort: 8081
```

Figure 5.5: Deployment of the Load-generator component

Same applied for the Kubernetes services I comment on a few attributes to add clarity to the Figure 5.6:

1. identifying object or controllers inside the Kubernetes domain
2. type of external service for more information see, 2.1
3. port where application is accessible inside running, container

[3]Definition of the image https://docs.docker.com/engine/reference/builder/

4. port where application is accessible from Service,
5. port where application is accessible from external Service.

```
apiVersion: v1
kind: Service (1)
metadata:
  name: my-quarkus-load-generator
spec:
  type: NodePort (3)
  ports:
  - port: 8081 (4)
    targetPort: 8081 (5)
    protocol: TCP
    name: http
    nodePort: 30002 (6)
```

Figure 5.6: Service of the Load-generator component

Chapter 6

Testing

The application has been tested in various ways at different levels. Starting from the lowest level of unit tests guaranteeing functionality and stability. Moreover, detecting of errors immediately though to system tests. Finally, a combination of created applications with the system Strimzi was tested by a marathon test. In other to be able to extend the app effectively without any errors, I have implemented a continues-integration pipeline, which checks essential dependencies and runs smoke tests.

6.1 Testing regression

Units of this system are considered, for instance, objects that are representing the whole record. In our case, it is the air-pollution object. The correct approach on how to achieve the state when we can claim that class has a 100% of method covered by tests is as follows. Firstly, we need to make sure that every value attribute is tested after the assignment. Additionally, we need to take care of arrangements and try every path that can be reachable in our code.

In our case, we need to test the behaviour of each component. This is considered as a module test of the specific component. We are testing how it reacts to the messages that we are sending. Unfortunately, two components can be tested in this way because the Exposer is strictly dependent on data, which are produced by Load-Generator and without them is not able to process it and do aggregation part. The Load-generator has two types of tests:

- **Module level** – we can imagine it as a test that will check the whole behaviour of class itself. In other words, in the test case, we create the object representation of the class and invoke a specific method.
- **System level** – in this case, we need to consider part to create the whole system with the Kafka cluster, zookeeper and the instance of Load-Generator, which uses streams with the specific name of the name using @Outcoming(„x“), where the x is the name of the stream. Here, we have a few dependencies to our system, for instance:
  - Debezium implementation of in-memory Apache Kafka cluster with the zookeeper for processing and elegant usage. Encapsulation of the complex system and create a simple interface, which can be described by two words as a Facade pattern. [16]

– Smallrye channels - used for catching the messages from the specific stream invoked by the Load-Generator. [1]

Furthermore, previously described stuff can be applied to other components. What is an exception is that on the system level, we are not able to verify it. I have discovered that if I want to test two components because the Exposer can not be tested as one unit on the System level. This is from a simple problem, because Exposer needs data from the Load Generator, and if the Load Generator does not exist, it is impossible to test it.

Last, the component is the React frontend application, and there is the smoke test, which just checks that the app can deploy, and we do not have a white screen. It is just starter before, verifying that google-api works and more.

6.2 Marathon test

The verification phase of this system is a little bit complicated. The first idea can be to check each message that is sent to the Strimzi and every single message that is retrieved. But we will simulate load equal to one hundred thousand records per second, therefore checking each particular message is impossible. The different ideas could be to check logs and the status of the whole application. In the Kubernetes world, we will just check pod, custom resource statuses, and metrics exported by Kafka. This approach can be effortlessly delivered compared to the first scenario.

Moreover, we need to take into consideration the following steps to create a pipeline with the following stages:

- **automation** – point can be covered by Junit5 framework
- **kubernetes client** – creating abstract Kubernetes client, which will encapsulate CLI client commands to use it in java code
- **reporting** – to be able to access this logs via top-level with the help of Jenkins [2] tool

Furthermore, we designed system tests from the perspective of marathon tests. The goal of these types of tests is found bugs in the product that can only be found in long term testing. One of the test cases is that we deploy a whole application with the Strimzi and periodically every 30 minutes check the status of all pods together with additional detailed information about Kafka resources. A load of generating messages can be parametrized by environment variables inside the load generator container. In general, these tests should run around one week to have some impact and informal value. Following figure 6.1 shows these steps.

6.3 Continuous Integrations

Martin Fowler, once said that is a software development practise where members of a team unite their work regularly, typically, each person integrates at least daily - leading to multiple integrations per day. Each sequence is verified by an automated build (including test) to detect integration errors as quickly as possible. Many companies find that this approach leads to significantly decreased integration difficulties and allows a team to develop cohesive software more swiftly.[17]

[1] Smallrye channels - https://smallrye.io/smallrye-reactive-messaging/#_channel
[2] CI/CD tool https://jenkins.io/

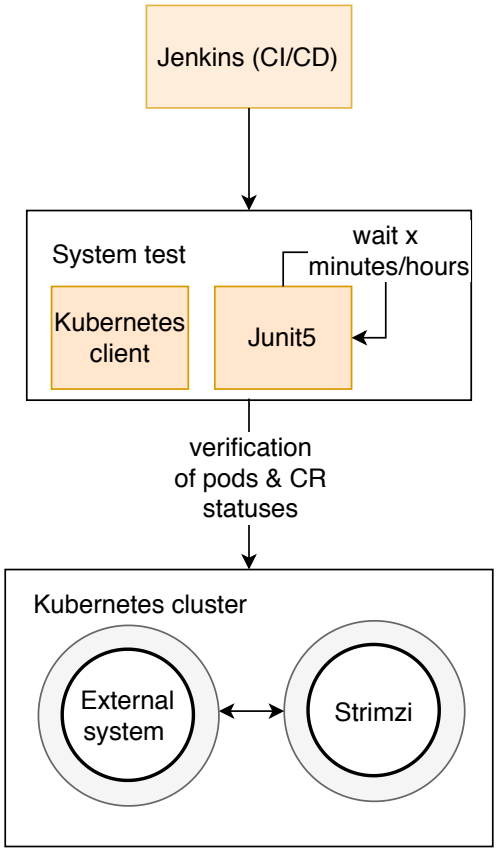


Figure 6.1: Process of the designed marathon test

Continuous Integration is considered as an essential part of the whole system for future work and the smooth development of new features. In my case, I have implemented continuous integration with the **Github actions**. The definition of the pipeline can be seen in the following Figure 6.2.

In this case, we can see that this is just pipeline that verify that application can be built with all type of dependencies with also transitive one. We can some of these attributes described in points to make it more clear:

1. name of the pipeline
2. triggers CI on every push or pull-request by some contributor. In other words, if we do some change to the Github repository, it will start the verification phase of the whole application
3. selected image, where the whole application will be tested. In different statements, an environment is selected for the testing; in this case, it is ubuntu-latest, which is the Debian distribution of the Linux.
4. Lastly, we have the steps, which must be in all the pipelines. In these steps, we sequentially write all tasks that have to be one like in this case, we installing needed dependencies, and finally, we build the projects Quarkus and then React.

```
name: simple-continous-integration (1)
on: [push, pull_request] (2)
jobs:
  build:
    runs-on: ubuntu-latest (3)
    steps: (4)
      - uses: actions/checkout@v1
      - name: Install Java Development Kit Version 11
        uses: actions/setup-java@v1
        with:
         java-version: 11.0.2
      - name: Install Maven
        run: sudo apt install maven
      - name: Build the Quarkus
        working-directory: ./Quarkus
        run: |
          mvn clean install -DskipTests
      - name: Install Nodejs & Npm
        run: |
          sudo apt install nodejs -y
          sudo apt install npm -y
          sudo npm install npm@latest -g
          sudo npm install nodejs@latest -g
          sudo nodejs -v
          sudo npm -v
      - name: Install & Build the React
        working-directory: ./React
        run: |
          sudo npm install -g
          sudo npm run-script build
```

Figure 6.2: Service of the Load-generator component

6.4 Experiments

The main aim of these experiments was to obtain information about the reliability of the system for specific purposes. What's more, due to the data provided by the load-generator. I have used few open-sourced tools as follows:

- **The Prometheus** [3] monitoring tool designed for the Kubernetes environment. Its main responsibility is to take all the metrics from the Kafka cluster and provide it to the Grafana.
- **The Grafana** [4] is a visualisation tool taking the data from the Prometheus we are able to see it an abstract way in the form of dashboards.

[3] Alerting toolkit provides data - https://prometheus.io/docs/introduction/overview/
[4] Monitoring tool, known for their beautiful dashboards - https://grafana.com/

In our case, we decided to do two types of testing. Firstly, we focused on the low load and deployment with one Kafka cluster, which contains three nodes of Kafka and three nodes of Zookeeper.

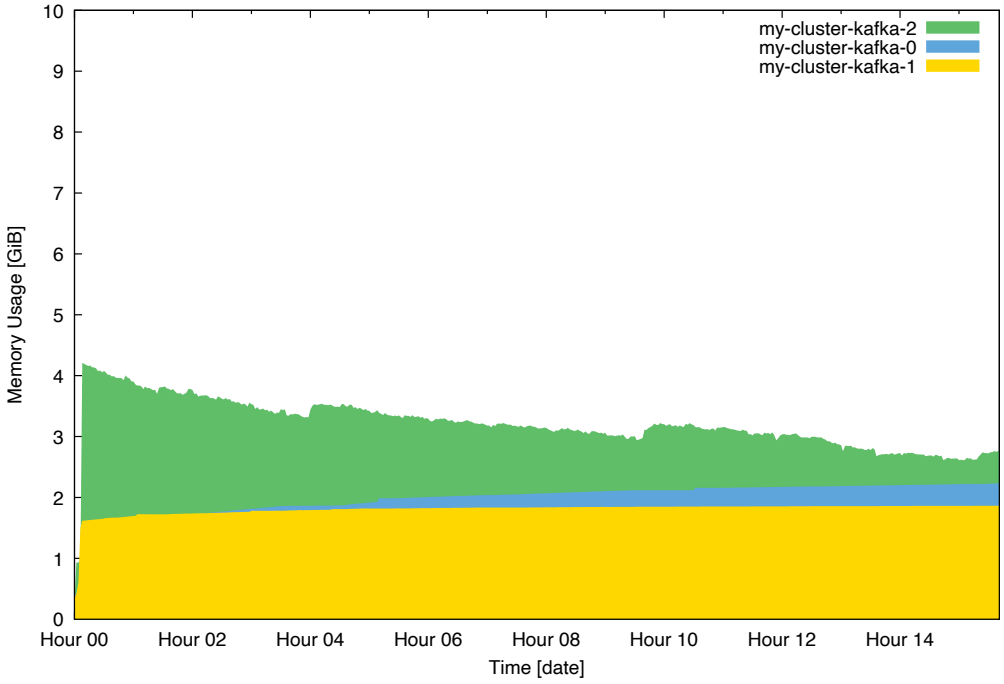


Figure 6.3: First Marathon test - metrics (memory-usage)

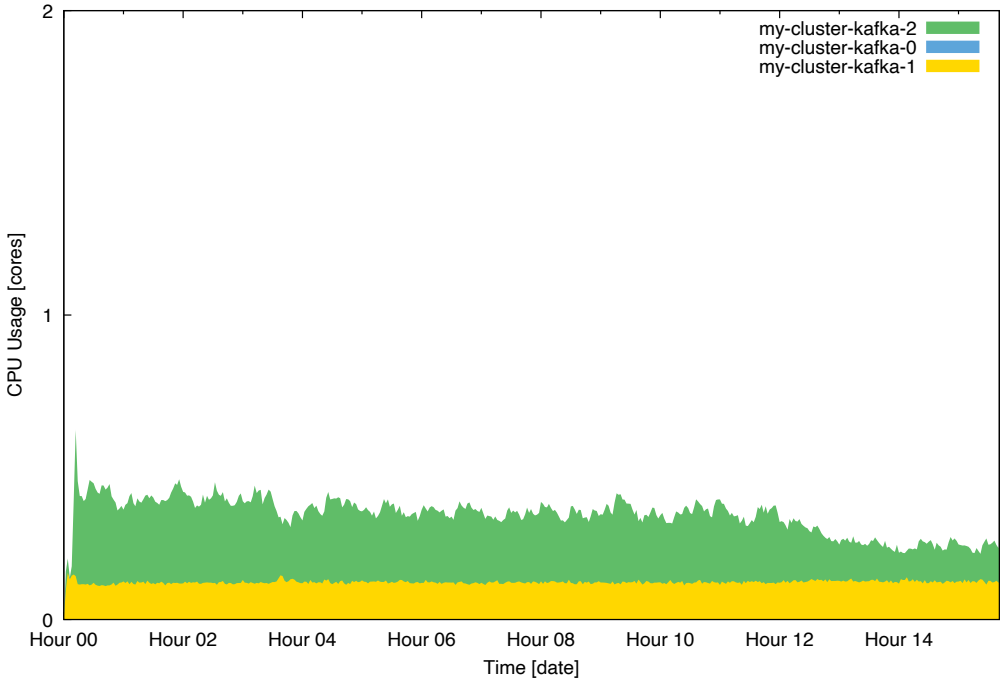


Figure 6.4: First Marathon test - metrics (cpu-usage)

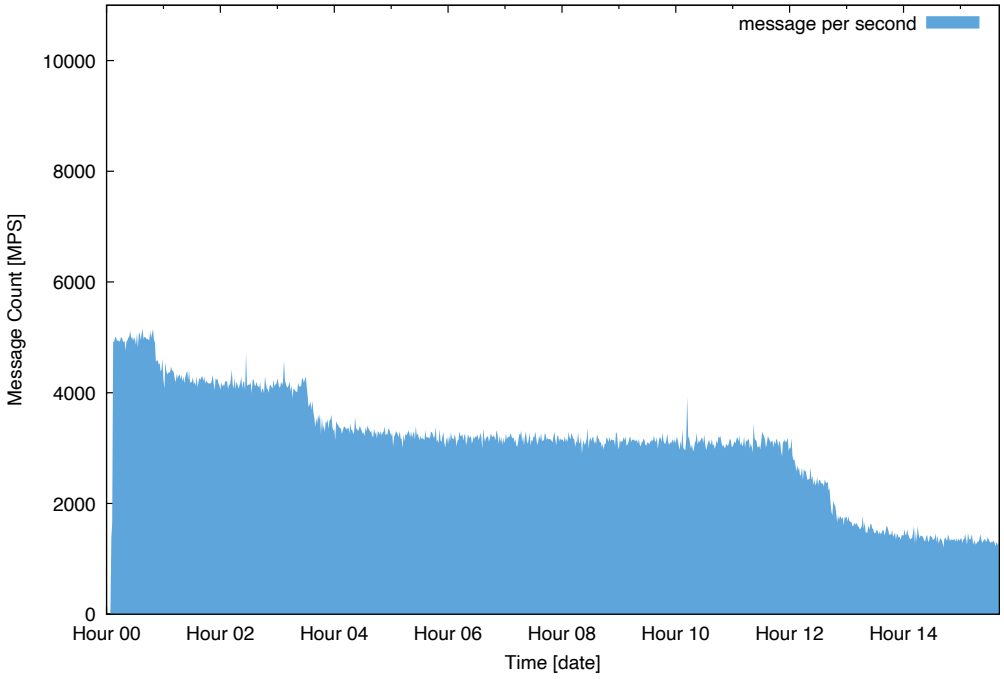


Figure 6.5: First Marathon test - metrics (message-per-second)

The test lasted approximately 16 hours, and it was found that there was no error in the system. We generate approximately five thousand messages per second, where each message has around 500 bytes. What was interesting to observe that after a while ratio of messages was at the lowest point about two thousand messages.

Also, we must take into consideration that message is a representation of the air pollution object, which is then shown to the user. This is the baseline experiment shows that the Strimzi can handle this load simultaneously for some period of time. The figure 6.3, shows that the system was ready for that load and nothing special happened.

Moreover, it was interesting to observe that we only produce and consume roughly from five thousand to ten thousand messages per second even when we set the counter of an environment variable to generate about one-hundred thousand messages per second. This situation could happen for various causes. For instance the network restriction, Quarkus Kafka client limitation and more.

The second and the heaviest were past about 14 hours with a load of sending and receiving ten-thousand messages. In the Figures 6.6, 6.7, 6.8 to compare with the result from previous experiment 6.3, 6.4, 6.5 are values more than doubled. The main reason for this is that Kafka saves records for one week by default, but the latest one Kafka saves in-memory to quick processing and immediate response. We have also checked other parameters as follows:

- **Network processor average idle in percentage** – does not goes below 90% for all previously mentioned experiments
- **Request handler average idle in percentage** – does not goes below 90% for all previously mentioned experiments

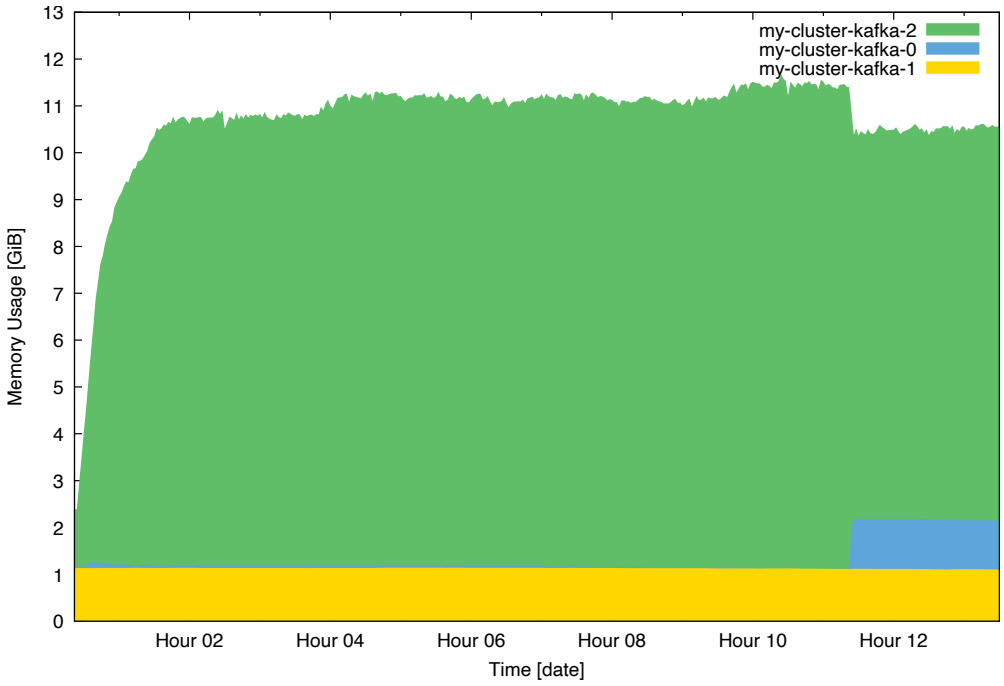


Figure 6.6: Second Marathon test - metrics (memory-usage)

- **Producer request rate** – always matched with the messages per second generated by Load-generator
- **Fetch request rate** – request rate, depends on the the Exposer, which consuming the messages
- **Byte rate** – calculated by the formula *message count * message size*
- **JVM** [5] **memory used**
- **CPU usage**

Furthermore, what is worth to mention that we also experiment with the count of the **Load-generator** and **Exposer** replicas, instances. For instance, we have deployed like fifty Pods of that application where all data were generated to the Kafka cluster and most of the time the whole cluster died cause of memory and CPU was not ready for such a load.

Based on the performed experiments, we have found that the Strimzi system is reliable. Meaning of reliability, in this case, means that the Strimzi was able to respond to client requests in the quick period of time, and there was also no critical error that would cause the system to be shuted down.

[5] Java virtual machine - https://www.geeksforgeeks.org/jvm-works-jvm-architecture/

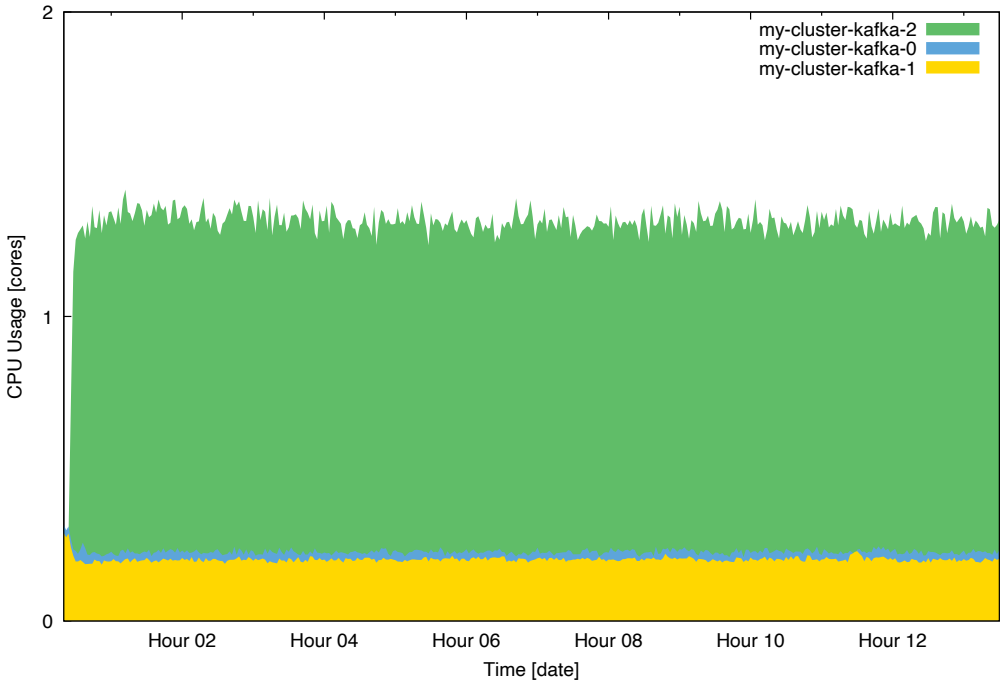


Figure 6.7: Second Marathon test - metrics (cpu-usage)

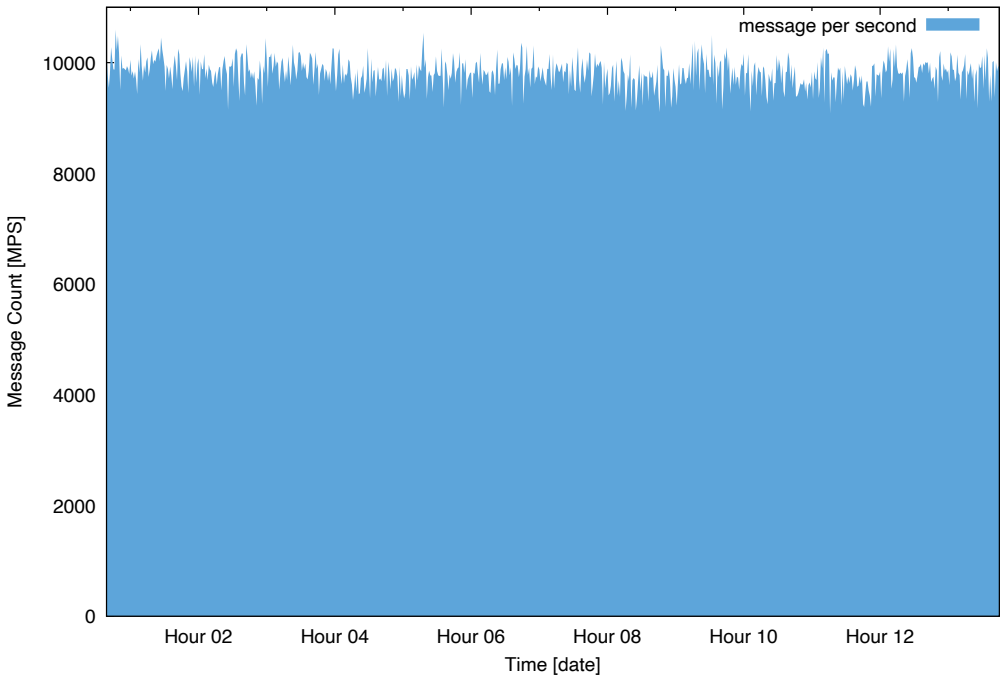


Figure 6.8: Second Marathon test - metrics (message-per-second)

Chapter 7

Conclusion

This thesis aimed to design, implement, and test the application, which demonstrates the user conditions. Moreover, the created app claims that the system Strimzi is reliable. We mainly focus on the backend of the created application based on the sub-atomic java Quarkus. Furthermore, we have focused on the clustering side, which was supported by the Kubernetes. Besides, we have learned and described the necessary technologies for the creation of this application.

Based on this knowledge, We have implemented the external system, which is used as a separate component in our project Strimzi for the load-testing more specifically for the long-term tests. What is worth to mentioning is that we also consulted the designed components with the Quarkus developers from the Redhat. Moreover, we have introduced the designed marathon test, as a is the verification of the system Strizmi. The designed application was tested on various levels. Starting from the units test, module test to the system level, where we have experimented with the marathon testing. In the end, we have also created a simple continuous integration by Github actions, which serves as a starting point to be able quickly to extend the application with quickly finding the bugs and detect bugs.

This work was shown at the community-central summit[1] at the RedHat, where presentation was performed. Secondly, we have created another presentation for the Open-house [2] event, which is organized every year for sharing of hot themes, cutting-edge technologies and more. Additionally, the article[3] was written to the Excel@FIT2020[4], which, in a nutshell, summarized all the substantial parts written in this work.

We probably all know the quote: „There is always room for improvement.“ My work is not an exception. I see the areas where it can be useful, for instance, in integration tests or marathon testing. We can actually characterize this practice as a pioneer in long-term testing. Fortunately, this work has a future in terms of extension, when we plan to use it and after that modify and extend it. For instance, to create a better and sophisticated verification of the system itself.

[1] In case of interest more can be found https://www.redhat.com/en/summit
[2] Redhat event, where are shown all hot themes - https://openhouse.redhat.com/cz/
[3] https://github.com/Godzillah/excel-article/blob/master/2020-DataProcessingWithStrimzi.pdf
[4] http://excel.fit.vutbr.cz/

Bibliography

[1] AUTHORS, K. *Kubernetes* [online]. 2019 [cit. 2019-11-10]. Available at: https://mapr.com/products/kubernetes/assets/k8s-logo.png.

[2] AUTHORS, O. *Graal Virtual Machine* [online]. 2020 [cit. 2020-04-20]. Available at: https://www.graalvm.org/.

[3] AUTHORS, O. *Open Java Development Kit system* [online]. 2020 [cit. 2020-04-20]. Available at: https://openjdk.java.net/.

[4] AUTHORS, T. B. *CDI description* [online]. 2019 [cit. 2019-12-04]. Available at: https://www.baeldung.com/java-ee-cdi#overview.

[5] AUTHORS, T. K. *Queuing and publish & subscribe model* [online]. 2019 [cit. 2019-16-10]. Available at: http://kafka.apache.org/documentation.html#intro_consumers.

[6] AUTHORS, T. K. *Custom resource* [online]. 2019 [cit. 2019-11-24]. Available at: https://medium.com/velotio-perspectives/extending-kubernetes-apis-with-custom-resource-definitions-crds-139c99ed3477.

[7] AUTHORS, T. K. *History* [online]. 2019 [cit. 2019-11-24]. Available at: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#going-back-in-time.

[8] AUTHORS, T. K. *Kube-controller-manager* [online]. 2019 [cit. 2019-11-09]. Available at: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/.

[9] AUTHORS, T. K. *Kubernetes components* [online]. 2019 [cit. 2019-12-06]. Available at: https://kubernetes.io/docs/concepts/overview/components/.

[10] AUTHORS, T. K. *Namespaces* [online]. 2019 [cit. 2019-11-10]. Available at: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[11] AUTHORS, T. K. *Operator* [online]. 2019 [cit. 2019-11-24]. Available at: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[12] AUTHORS, T. K. *Service* [online]. 2019 [cit. 2019-11-10]. Available at: https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types.

[13] AUTHORS, T. S. *Topic operator* [online]. 2019 [cit. 2019-11-24]. Available at: https://strimzi.io/docs/latest/.

[14] Authors, T. S. *Strimzi system* [online]. 2020 [cit. 2020-03-20]. Available at: https://strimzi.io/.

[15] authors w3schools. *Basic concepts of React javascript library* [online]. 2017 [cit. 2019-12-23]. Available at: https://www.w3schools.com/react/react_lifecycle.asp.

[16] Community, D. *Debezium* [online]. 2020 [cit. 2020-02-14]. Available at: https://debezium.io/.

[17] Fowler, M. *Continuos Integration* [online]. 2020 [cit. 2020-02-16]. Available at: https://martinfowler.com/articles/continuousIntegration.html.

[18] Guru99. *Load Testing Tutorial: What is? How to? (with Examples)* [online]. 2019 [cit. 2019-11-09]. Available at: https://www.guru99.com/load-testing-tutorial.html.

[19] IconArchive.com. *Computer* [online]. 2019 [cit. 2019-11-10]. Available at: http://www.iconarchive.com/show/3d-bluefx-desktop-icons-by-wallpaperfx/Monitor-icon.html.

[20] Inc., D. *Docker* [online]. 2016 [cit. 2019-11-10]. Available at: https://s3-us-west-2.amazonaws.com/com-netuitive-app-usw2-public/wp-content/uploads/2016/06/small_v-trans.png.

[21] Inc, F. *React – A JavaScript library for building user interfaces* [online]. [Online; visited 2019/11/03]. Available at: https://reactjs.org/docs/getting-started.html.

[22] Inc., F. *Fundamentals of React* [online]. 2017 [cit. 2019-12-23]. Available at: https://reactjs.org/docs/.

[23] MimasTech. *KVM* [online]. 2016 [cit. 2019-11-10]. Available at: http://www.mimastech.com/wp-content/uploads/2016/03/kvm-logo.png.

[24] Redhat. *Quarkus, Supersonic & Subatomic java* [online]. [Online; visited 2019/09/15]. Available at: https://quarkus.io/guides/.