# Implementation of an Enterprise Service Bus with OpenShift and Camel

Ing. Thomas Herzog B.Sc

## MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Software Engineering

in Hagenberg

im Juni 2018

# Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 1, 2018

Ing. Thomas Herzog B.Sc

# Contents

# Preface

# Abstract

This should be a 1-page (maximum) summary of your work in English.

# Chapter 1

# Introduction

## 1.1 Motivation

Large enterprises work with several independent applications, where each application covers an aspect of a business of the enterprise. In general, these applications are from different vendors, implemented in different programming languages and with their own life cycle management. To provide a business value to the enterprise, these applications are connected via a network and they contribute to a business workflow. The applications have to interexchange data, which is commonly represented in different data formats and versions. This leads to a highly heterogeneous network of applications, which is very hard to maintain.

The major challenge of an IT department is the integration of independent applications into the enterprise application environment. The concept of Enterprise Application Integration (EAI) provides patterns, which help to define a process for the integration of applications into a heterogeneous enterprise application environment. One of these patterns is the Enterprise Service Bus (ESB), which is widely used in the industry [HW08].

Often the term ESB application is used to refer to an ESB, which integrates internal and external hosted applications. But an ESB is a software architectural model, rather than an application. The term could have been established by the usage of middleware such as JBoss Fuse, which provides tooling to integrate applications into an ESB [Red18a]. JBoss Fuse is based on the JBoss Enterprise Application Platform (JBoss EAP), where the applications are integrated in a existing runtime environment.

With the upcoming of cloud solutions such as Platform as a Service (PaaS) it is now possible to move the platform from a dedicated environment to a cloud environment, where each integration service has its own runtime environment rather than joining an existing runtime environment. The concept of Integration Platform as a Service (IPaaS) relies on top of PaaS and enhances a common PaaS solution with the Integration features needed by EAI [DG15; Liu+15].

Thus, enterprises can reduce the effort in implementing and maintaining an ESB, integrating applications into the ESB and reducing the costs of an ESB by using a consumption based pricing model.

## 1.2 Objectives

This thesis aims to implement an ESB on Openshift PaaS [Red18b]. Commonly an ESB is implemented with the help of middleware such as JBoss Fuse, which is based on the JBoss EAP. The concepts of PaaS and IPaaS are in general new to the industry, which commonly hosts their integration services in their own data centers, due to the lack of trust for cloud solutions and knowledge about the new approaches such as microservice architecture.

A main focus of this thesis is how applications internal and external can be integrated and managed in the PaaS solution Openshift with the ESB pattern. Before implementing an ESB in a PaaS solution such as Openshift, its necessary to understand the new concepts such as Infrastructure as a Service (IaaS), or containerization with Docker, which are covered in the following chapters. The microservice approach and cloud solutions are becoming more important for the software industry. For instance, Red Hat is currently moving its ESB middleware JBoss Fuse to the cloud, where JBoss Fuse will fully rely on Openshift, and the integration services have to be implemented as microservices. This has huge impact on Red Hats customers, who are used to JBoss Fuse on top of JBoss EAP.

This thesis was commissioned by the company Gepardec IT Services GmbH, a company that is working in the area of Java Enterprise and cloud development. The migration from a monolithic ESB to a microservice structured ESB, which is hosted in a PaaS environment, is a major concern for them. The migration from a monolithic ESB to a microservice structured ESB will be a major challenge for their customers, because microservice architecture and cloud solutions are mostly new to them.

Over the past years, a huge technology dept has been produced by the industry, due to the monolithic architecture and little refactoring work on their applications and hosting infrastructure. It will be hard for them to reduce the produced technology dept, which they will have to, to keep competitive. Gepardec sees a lot of potential for their business and their customers in this new approach of implementing and hosting an ESB.

# Chapter 2

# Infrastructure as Code

Infrastructure as Code (IaC) is a concept to automate system creation and change management with techniques from software development. Systems are defined in a Domain Specific Language (DSL), which gets interpreted by a tool, which creates an instance of the system or applies changes to it. IaC defines predefined, repeatable routines for managing systems [Kie16]. IaC descriptions are called templates, cookbooks, recipes or playbooks, depending on the tool. In the further course, the IaC definitions will be called templates. The DSL allows to define resources of a system such as network, storage and routing descriptively in a template. The DSL abstracts the developer from system specific settings and provides a way to define the system with as little configuration as possible. The term system is used as a general description. In the context of IaC, a system can be anything which can be described via a DSL.

## 2.1   The Need for Infrastructure as Code

In the so called iron age, the IT systems were bound the physical hardware and the setup of such a system and its change management were a long term, complex and error prone process. These days, we call such systems legacy systems. In the cloud age, the IT systems are decoupled from the physical hardware and in the case of PaaS they are even decoupled from the operating system [Kie16]. The IT systems are decoupled from the physical hardware and operating system, due to the fact, that cloud providers cannot allow their customer to tamper with the underlying system and hardware. In general, the hardware resources provided by a cloud provider are shared by multiple customers.

With IaC it is possible to work with so called Dynamic Infrastructure Platforms, which provide computing resources, where the developers are completely abstracted from the underlying system. Dynamic infrastructure platforms have the characteristic to be programmable, are available on-demand and provide self service mechanisms, therefore we need IaC to work with such infrastructures [Kie16]. Systems deployed on a dynamic infrastructure platform are flexible, consistent, automated and reproducible.

Enterprises which stuck to legacy systems face the problem that technology nimble competitors can work with their infrastructures more efficiently, and therefore can demand lower prices from their customers. This is due to the IaC principles discussed in Section 2.2 on the following page. Over a short period of time, enterprises will have to move to IaC and away from their legacy systems to stay competitive. The transition process could be challenging for an enterprise, because they lose control over the physical hardware and maybe also over the operating system. Maintaining legacy systems has the effect that someone is close to the system and almost everything is done manually. IaC has the goal to automate almost everything, which requires trust for the cloud providers, who provide the computing resources and the tooling, which provides the automation. A well known problem, which enterprise will face, is the so called Automation Fear Spiral, which is shown in Figure 2.1.
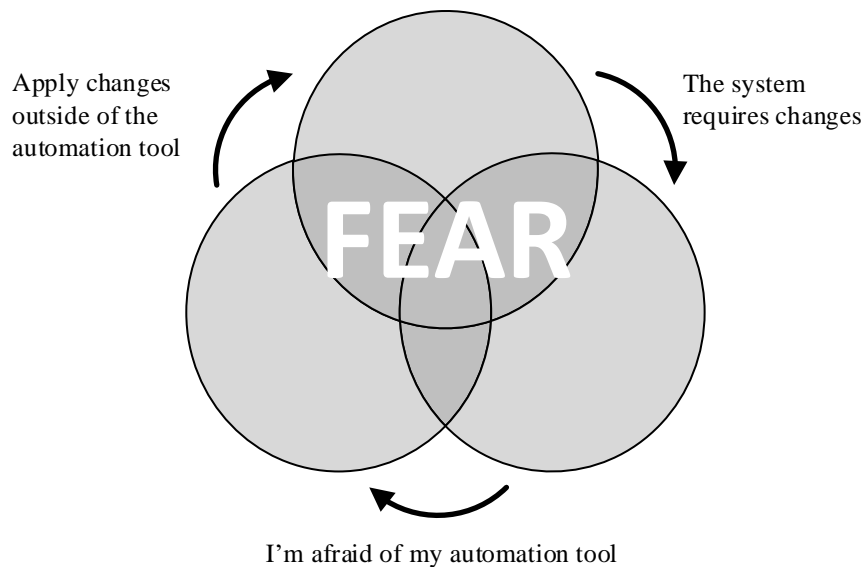


**Figure 2.1:** Automation Fear Spiral

Because of no trust for the automation, changes are applied manually to the systems and outside the defined automation process. If the system is reproduced, definitions may be missing in the templates, which leads to an inconsistent system. Therefore, enterprises have to break this spiral to fully profit from IaC [Kie16].

When enterprises have moved their legacy systems to IaC, they can not only manage their systems faster, they also can profit from the principles of IaC as discussed in Section 2.2 on the following page. With IaC, systems are less complicated to manage, changes can be applied without fear, and the systems can easily be moved between environments. This provides the enterprises with more space to maneuver, systems can become more complex but still easy to manage, the systems can be defined and

created faster which could lower costs.

## 2.2 Principles of Infrastructure as Code

The principles of IaC solve the problems of systems of the iron age. In the iron age the creation and maintenance of systems were a long, complicated and error prone process which consumed a lot of resources and time. With the decoupling of the physical hardware from the system, the creation and maintenance of the system has become simple, due to the IaC DSL and tooling.

### 2.2.1 Infrastructures are Reproducible

With IaC, systems are easy reproducible. It is possible to reproduce the whole infrastructure or parts of it effortlessly. Effortless means, that no tweaks have to be made to the templates or during the reproduction process and there is no need for a long term decision process about what has to be reproduced and how to reproduce it. To be able to reproduce system effortlessly is powerful, because it can be done automatically, consistently and with less risk of failures [Kie16]. The reproducibility of a system is based on reusable templates which provide the possibility to define parameters, which are set for the different environments as shown in Figure 2.2.
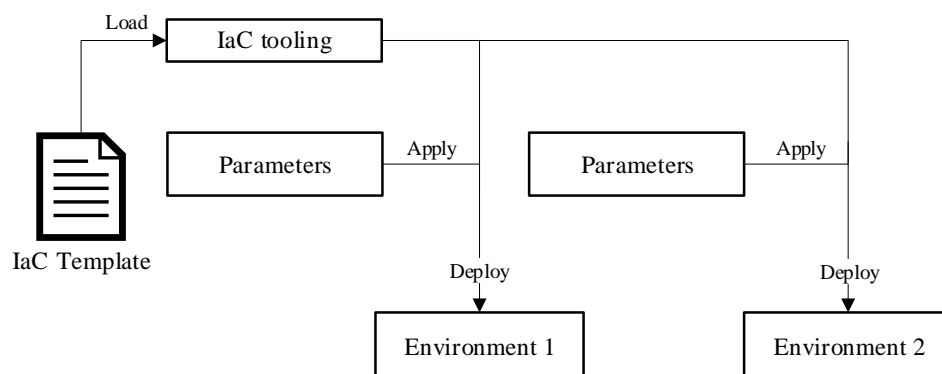


**Figure 2.2:** Schema of a parametrized infrastructure deployment

### 2.2.2 Infrastructures are Disposable

Another benefit of IaC is that systems are disposable. Disposable means, that systems can be easily destroyed and recreated. Changes made to the templates of a system does not have to be applied on an existing system, but can be applied by destroying and recreating the system. An requirement for a disposable system is, that it is understood that systems will always change. Other systems relying on a disposable system need to address that the system could change at any time. Systems

must not fail because a disposable system disappears and reappears again because of an redeployment [Kie16].

### 2.2.3   Infrastructures are Consistent

Systems managed with IaC are consistent, because they are defined via a template and all instances are an instance of the template, with the little configuration differences defined by parameters. As long as the system changes are managed by IaC, the system will stay consistent, and the automation process can be trusted.

In Listing 1 an example for an IaC template is shown, which defines a Docker Compose service infrastructure for hosting a Wildfly server instance [Doc18d; Red17]. This system can consistently be reproduced on any environment supporting Docker, Docker Compose and providing values for the defined parameters.

```
1  version: "2.1"
2  services:
3    wildfly:
4        container_name: wildfly
5        image: wildfly:latest
6        ports:
7            - "${EXPOSED_PORT}:8080"
8        environment:
9            - "POSTGRES_DB_URL=${POSTGRES_DB_URL}"
10           - "POSTGRES_DB_NAME=${POSTGRES_DB_NAME}"
11           - "POSTGRES_USER=${POSTGRES_USER}"
12           - "POSTGRES_PASSWORD=${POSTGRES_PASSWORD}"
```

**Listing 1:** Example for an IaC template for Docker Compose

### 2.2.4   Actions are Repeatable

Building reproducible systems, means that any action applied to the system should be repeatable. Without repeatability, the automation cannot be trusted and systems wouldn't be reproducible. An instance of a system in another environment should be equal to any other system instance, except for the configurations defined by parameters. If this is not the case, then a system is not reproducible, because it will have become inconsistent [Kie16].

IaC is a concept which makes it very easy to deal with systems in the cloud age. Enterprises can make use of IaC to move their legacy systems to the cloud, where they can profit from the principles of IaC. Nevertheless, before an enterprise can profit from IaC, it has to apply clear structures to their development process, as well as sticking to the principles of consistency and repeatability. For experienced administrators, who are used to maintain systems manually, it could sometimes be hard to understand why they are not supposed to perform any actions on the system manually anymore, nevertheless that a manual change could be performed faster.

Being capable to reproduce a system at any time with no effort, or applying changes on an existing system in a predefined and consistent manner, makes enterprises very flexible and fast. Enterprises will not have to fear future changes in requirements and technologies of their systems anymore.

# Chapter 3

# Containerization with Docker

Docker is a tool for creating, provisioning and interacting with Linux Containers (LXC) [Doc18f; Lin18a]. LXC are a lightweight version of virtualization, which does not have the resource impact of a full virtualization such as Operating System (OS) virtualization. The differences of LXC and a Virtual Machine (VM) are covered in Section 3.3. Docker has become very popular over the past years, due to the fact, that it made it possible to easily work with LXC. Docker relies strongly on the principles of IaC which has been discussed in Chapter 2. When using Docker, Linux Containers are often referred to as Docker Containers.

Containerization is a key factor when hosting applications in the cloud, because the applications are normally packaged in images and run as containers on the cloud platform. Containerization provides features for a fast, effortless and consistent way of running applications in the cloud, which is discussed in the following Section 3.1.

## 3.1  The need for Containerization

Containerization is a key factor for cloud platforms such as PaaS, where each application runs in its own isolated environment, called a container. A container is an instance of an image, which represents the initial state of an application. A VM represents a full blown OS, where the OS provides a kernel, which is emulated on the host OS by the Hypervisor. A Hypervisor is a software which can create, run and manage VMs. A container uses the kernel provided by the host OS and therefore there is no need for an emulation. A container does not represent a full blown OS, but still provides features normally provided by an OS such a networking and storage [Sch14].

Containers are faster to create, to deploy and easier to manage compared to VMs. Nevertheless, cloud platforms use virtualization for managing their infrastructure, where the containers run on the provisioned VMs. The usage of containers compared to the usage of VMs can reduce costs for hosting applications. Enterprises can profit from hosting their applications of containers in several ways. Applications hosted in containers need lees resources than applications hosted in VMs, because

there is no virtualized OS and no need for kernel emulation. The creation, deployment and startup of containers are faster, because only the isolated process needs to be started and not a full blown OS. Docker is well supported by Integrated Development Environments (IDEs), which provide support for creating Docker Image definitions (Dockerfiles) and provisioning of Docker Containers on a local or remote environment [Doc18b].

When enterprises have applied IaC to their infrastructure, then the next logical step is to integrate their applications into IaC as well. Applications hosted in containers profit from the IaC principles immutability, reproducibility, repeatability and consistency. Therefore, Docker strongly relies on IaC and provides tooling for automating creation and provisioning of Docker Containers, which is used by PaaS platforms such as Openshift. With Docker, developers define the hosting environment for their applications and not system administrators anymore. Nevertheless, developers can profit from the deep Linux knowledge of system administrators, to define the Docker Images efficiently, to keep them small and secure. The following Section 3.2 will give an overview of the Docker technology, its architecture and artifacts.

## 3.2 Docker

This section covers Docker, which is the most popular tool to work with LXC. Docker is open source but also provides an enterprise support. The core part of the Docker technology is the Docker Engine, which is discussed in Section 3.2.1. The Docker Engine is the part of the Docker technology that actually runs the containers. The Docker Images are managed in a so called Docker Registry, which is a repository for Docker Images. The most popular Docker Registry is Docker Hub, which is a free service, where anyone can provides Docker Images [Doc18c].

### 3.2.1 Docker Engine

Figure 3.1 illustrates the Docker Engine architecture hosted on a Linux OS. The Docker Engine is build by layers, where each layer communicates with the layer beneath.
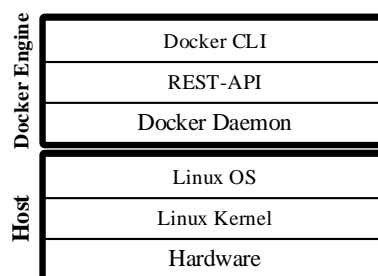


**Figure 3.1:** Docker Engine architecture

The Docker Engine was initially designed for LXC exclusively but has been ported to Windows. Docker Images and Containers created for Windows OS are not supported on a Linux OS and visa versa. The Docker Images and Containers for a Windows OS differ from those for a Linux OS, but the principles of Docker Images and Docker Containers are the same.

**Docker Daemon**

The Docker Daemon represents the background process, which creates, runs and manages the Docker Containers on the Docker Host, similar to a VM Hypervisor. The Docker Daemon strongly depends on the kernel of the host OS, therefore incompatibilities could cause the Docker Daemon to fail functioning. The communication with the Docker Daemon is performed via a REST-API, because the Docker Engine is designed as a server client architecture.

**REST-API**

The REST-API can be exposed via a Unix socket or a network interface, depending on the configuration of the Docker Daemon. If the REST-API is exposed via a network interface, then it is recommended to secure the connection with client certificate authentication. If the Docker Engine and the Docker Client are located on the same host, then commonly the REST-API is exposed via a Unix socket and does not need any special security.

**Docker Command Line interface**

The Docker Engine provides a Docker Command Line Interface (CLI) for interacting with the Docker Daemon via a Linux shell. The Docker CLI itself communicates with the Docker Daemon via the exposed REST-API. This is the most common way to interact with a Docker Daemon. The Docker CLI provides commands for creating Docker Images and Containers and for provisioning the Docker Containers on the Docker Host.

**Docker Images**

Docker Images are defined via Dockerfiles, which contain instructions how to build the Docker Image. A Docker Image consists of layers, where each layer represents a state of the file system, produced by a Dockerfile instruction. Each layer is immutable and any change on the file system produces a new layer. Docker Images are hierarchical and can inherit from another Docker Image, which is then called base image. Docker Images support only single inheritance and the base image is defined via the *FROM* instruction as the first instruction in the Dockerfile. Docker Image names have the structure *[namespace]/[name]:[version]* e.g. *library/openjdk:8-alpine*.

**Docker Containers**

A Docker Container is an instance of a Docker Image, where a new layer is appended, which contains all changes made on the file system by the running process within

the Docker Container. When the Docker Container is deleted, then the appended
layer gets deleted as well and all made changes on the file system are lost. A Docker
Container keeps running as long as the contained foreground process is running.
Without a foreground process the Docker Container stops immediately after it was
started. The process running in the Docker Container is isolated from other pro-
cesses, as well is the file system, the process has access to.

### 3.2.2   Docker Architecture

The Figure 3.2 illustrates the Docker architecture, which is a client server architec-
ture. The design as a client server architecture is the reason why the communication
to the Docker Daemon is performed via the provided REST-API. The Docker Client
communicates with the Docker Daemon via the Docker CLI, where the Docker Client
can be located on a remote host or on the Docker Host. The Docker Host hosts the
Docker Engine, which exposes the REST-API the Docker Client connects to. The
Docker Engine managed the Docker Images and Containers located on the Docker
Host. The Docker Engine can pull Docker Images from a remote Docker Registry, if
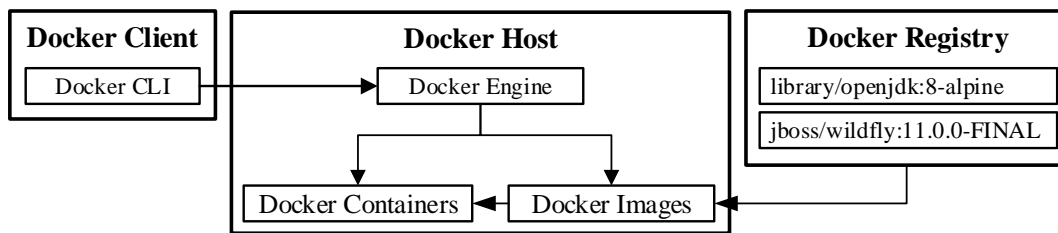a registry has been registered.

**Figure 3.2:** Docker Architecture

### 3.2.3   Docker Machine

Docker Machine is a tool for managing local or remote Docker Hosts [Doc18a]. With
Docker Machine an administrator can manage multiple Docker Hosts from a main
server, without the need to connect to the Docker Host via secure shell (SSH). The
Docker Machine CLI provides all commands necessary for managing Docker Hosts.
Docker Engine provisions Docker Containers on a Docker Host and Docker Machine
provisions Docker Hosts, in particular Docker Engines installed on docker Hosts.
With Docker Machines a network of Docker Hosts can be managed, which is used
by cloud platforms such as Openshift to manage Docker Engines on the nodes within
the Openshift cluster.

## 3.3 Virtualization vs. Containerization

Before LXC the industry made heavy use of operating system (OS) virtualization to isolate their environments and applications. A VM is managed by a Hypervisor, which is software, which can create, run and manage VMs. The VM provides resources such as network and storage for the application, which is managed by the virtualized OS. Nevertheless, an VM represents a full blown OS, which itself has a resource need which adds to the resource needs of the hosted application. LXC on the other hand are a kernel technology, which provides resources such as network and storage to the application as well, but without the need of virtualized OS.

### 3.3.1 Virtual Machines

A Virtual Machine is an instance of a Virtual Machine Image (VMI), which is managed by a Virtual Machine Monitor (VMM), which is also referred to as the Hypervisor. The actual difference between a VMM and a Hypervisor is where the software is installed on. If the software is directly installed on the Hardware, then the software is called a Hypervisor, if its installed on the Host OS then its called a VMM. The VM abstracts an Guest OS from the Host OS, in particular from the underlying hardware. A VM contained Guest OS is not bound to the underlying hardware, because the Hypervisor performs a kernel emulation, which allows to virtualize any Guest OS on any hardware, if the hypervisor supports it. The following Figure 3.3 illustrates the architecture of a virtualization system.
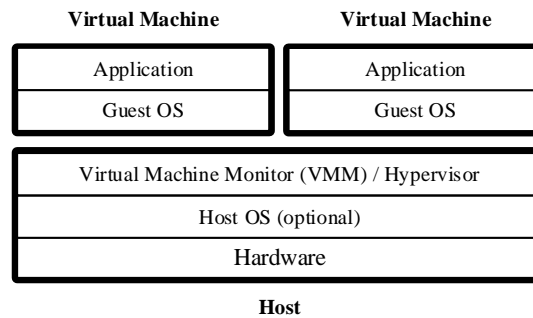
**Figure 3.3:** Architecture of virtualized applications

Glauber Costa's started the abstract of his talk at the LinuxCon 2012 with the humorous note *"I once heard that Hypervisors are the living proof of operating system's incompetence"*. With this note he expressed that OS weren't able to provide proper isolation for applications and therefore the industry started to provide an OS instance for each application [Cos12]. This has been overcome with the upcoming of LXC, which provide the proper isolation of applications on the same OS, which made the need for an OS instance for each application obsolete.

### 3.3.2   Linux Container

The upcoming of LXC has eliminated the shortcoming to not be able to isolate applications properly of the Linux OS, which lead to using OS virtualization to isolate applications. LXC provide the feature of isolating applications running on the same OS, without the need of a kernel and hardware emulation as it is done with OS virtualization. As illustrated in Figure 3.4, the application process, binaries and libraries are bundled into the container and are isolated from other containers. Each container gets a portion o the global resources such as CPU cycles and memory assigned and cannot consume more as it has been assigned to. Without LXC it is possible that one process takes over the system resources and other processes get into state of starvation, which lead to need of OS virtualization.
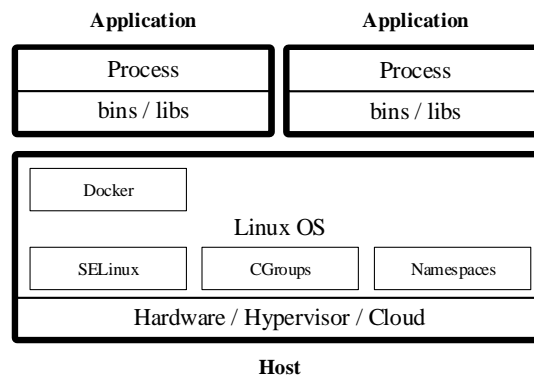


**Figure 3.4:** Architecture of containerized applications

The two most important kernel features underlying LXC are *Cgroups* and *Namespaces*. These two kernel features provide the resource control and isolation needed for application isolation and prevention of process starvation.

#### Cgroups

Cgroups stands for control groups and Cgroups provide the ability to aggregate processes, their child processes and threads within theses processes to groups managed in a tree structure. Each group gets a portion of the global resources such as CPU time, memory, I/O and network assigned, where its guaranteed that a group and its managed processes cannot consume more resources as the group has been assigned to. Each application hosted in a container is assigned to a group, where an application cannot steal resources from another application anymore, because the resource assignments of an group managed by Cgroups prevents this from happening [Cor14; Heo15; Men18].

#### Namepsaces

Cgroups manage how many resources can be used by processes in a group and

namespaces manage the view of the system to processes. A container is managed in a namespace and therefore it has a limited view of the system such as networks and Process IDs (PIDs), depending on the configuration of the namespace the container is part of. Namespaces are a fundamental concept of LXC, and namespaces provide the isolation of a container [Cor14; Lin18b].

Docker has made the usage of LXC simple, but it is very hard to maintain a large set of Docker Containers (>100) via the Docker CLI, or to implement and maintain a cluster of Docker Hosts with Docker Machine. To much would have to be scripted manually, which would fast become very hard to maintain. Additionally, Docker does not provide any workflow for deployment and scaling of Docker Containers, and also does not ensure that a desired state of the containers is met. For a local development or a small set of containers the Docker CLI, Docker Compose and Docker Machine are suitable, but when it comes to large dynamic infrastructures with a large set of Docker Containers to maintain, then container orchestration platforms like Kubernetes, which is discussed in Chapter 4 on the next page, will have to be used [Doc18e; Kub18d].

# Chapter 4

# Container as a Service with Kubernetes

Container as a Service (CaaS) is a term introduced by cloud providers, which provide a cloud based on demand container environment. But CaaS is more then just an on demand container environment like Docker, it provides orchestration and monitoring tooling for containers, and additionally CaaS is considered to be a model for IT organizations and developers how they can build, ship and run their applications anywhere. There are multiple CaaS providers on the market, but the most popular CaaS providers are Azure Container Service, Amazon Elastic Container Service for Kubernetes (Amazon EKS) and Google Kubernetes Engine, where they bring in their own flavor of CaaS but all of them use Kubernetes beneath [Ama18; Goo18; Kub18d; Mic18].

Kubernetes is a container orchestration platform for automating deployments, scaling and operation of containers across a Kubernetes Cluster of Kubernetes Worker-Nodes. Kubernetes has been invented by Google and is open source since 2015 and managed by the Cloud Native Computing Foundation, where the Cloud Native Computing Foundation is under the umbrella of the Linux Foundation. Kubernetes has become the most popular container orchestration platform on the market and is used by many CaaS and PaaS providers [Clo18].

## 4.1 The need for Container as a Service

Enterprises and developers are facing the need to dynamically apply to workloads and to roll out new version of their services fast. For applying dynamically to workloads a dynamic infrastructure is necessary to scale services up if the workload increases and to scale services down when the workload decreases, which is non trivial to be handled manually. Rolling out new versions requires a well defined workflow which specifies the roll out behavior, which also is non trivial to handle. For such uses cases a container orchestration platform like Kubernetes can be used, which provides workflows for roll out and support for scaling containers along with many other features. Kubernetes makes it possible to effortlessly manage complex service infrastructures, service scaling and the roll out of services. Thus, complex service infrastructures become simple to implement and manage.

Kubernetes uses IaC, which has been discussed in Chapter 2 on page 3, and therefore provides all of the principles of IaC as discussed in Section 2.2 on page 5. Kubernetes provides a DSL, which allows to specify the desired state of the Kubernetes Cluster such as running containers, container replicas and provided container resources such as RAM, CPU and network. Kubernetes automatically ensures that the state of the Kubernetes Cluster meets its specification. Thus, the developers have only the need to specify the desired state of their Kubernetes Cluster. Kubernetes provides enterprises an infrastructure for their services, which is effortlessly to specify and maintain, because of the automation tooling provided by Kubernetes. This makes it easy to modify the infrastructure at any time, which allows enterprises to apply fast to new requirements.

## 4.2   Kubernetes

Kubernetes is a platform to orchestrate containers in a cluster, where the Kubernetes Cluster-Nodes can be placed in the cloud or on a dedicated servers. Kubernetes is designed as a client server architecture and a master slave architecture. One node in the Kubernetes Cluster acts as the Kubernetes Master, which is discussed in Section 4.2.2 on page 18, and the other nodes in the Kubernetes Cluster act as the Kubernetes Workers, which are discussed in Section 4.2.3 on page 19. The Figure 4.1 illustrates the architecture of a Kubernetes Cluster.



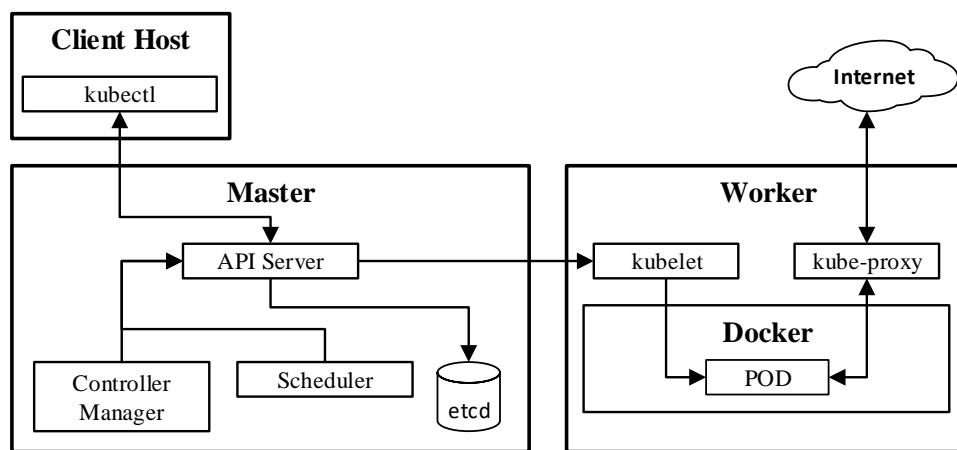**Figure 4.1:** Architecture of a Kubernetes Cluster

### 4.2.1   Kubernetes Objects

Kubernetes Objects are persistent objects in the Kubernetes system and the Kubernetes Objects describe the state of the Kubernetes Cluster. The Kubernetes Cluster

ensures that the state of the cluster meets the state specified by the Kubernetes Objects. The developers don't have to manually perform actions in the Kubernetes Cluster, they just have to modify the specification of the state of the Kubernetes Cluster and the Kubernetes Clusters itself will ensure that the new state is applied on the Kubernetes Cluster. The following sections will describe some of the common used Kubernetes Objects. The overview of all Kubernetes Objects is covered by the Kubernetes API reference documentation [Kub18a].

### Pod

A Pod is a group of one or more containers which are managed together. A Pod specification contains the specification for each container in the group. All of the containers of a Pod are always scheduled on the same Kubernetes Worker and will be deployed, started and stopped as a single unit. In a pre-container world all of the applications represented by the containers would have been hosted on the same physical machine. A Pod allows to bundle containers together which are acting as a single service, for instance a web application container with a caching container [Kub18c].

### Service

A service is an abstraction which defines a set of Pods and policies how to access them. The connection to the Pod via the service abstraction is handled by the Kubernetes Proxy. The service abstraction is necessary because a Pod can be hosted on any Kubernetes Worker within the Kubernetes Cluster, and the Pod will therefore get a random IP assigned which makes it impossible to address the Pod directly. If multiple replicas of a Pod are running, then the service will connect to a Pod of the replica set, depending on the chosen algorithm [Kub18e].

### Secret

A secret is an abstraction to manage sensitive data, which is consumed by containers. A secret holds sensitive data and hides it behind a name. The secret can be referenced by a container specification by its name. A referenced secret will be injected into the container either as a environment variable or a file. Developers reference secrets in the specifications by their name, and only the referencing containers can access the sensitive data the secret holds.

### ConfigMap

A configuration map is similar to a secret, but is intended to hold non sensitive data. A configuration map is meant to hold configurations such as logging configuration, which is consumed by containers. Configuration map also hide the data behind a name which can be referenced by a specification, and can they can be injected into containers the same way as secrets are. Configurations can be replaced during the container is up and and will be re-injected when the container restarts.

### 4.2.2 Kubernetes Master

The Kubernetes Master is the master node in the Kubernetes Cluster. It is responsible for managing the Kubernetes Worker-Nodes and containers running on those nodes. The Kubernetes Master exposes a REST-API via the clients can interact with the cluster. The node hosting the Kubernetes Master should be exclusively for the Kubernetes Master. The following sections briefly introduce the Kubernetes Master-Components, which are responsible for managing the Kubernetes Cluster [Kub18b].

#### Kubernetes CLI (kubectl)

Kubectl is the CLI of Kubernetes, which provides an interface to manage the Kubernetes Cluster and to manage Pods running on the Kubernetes Worker-Nodes. Kubectl is similar to the Docker CLI, but does not support direct interacting with Docker. Kubectl interacts with the Kubernetes Cluster via a REST-API exposed by the Kubernetes Master API-Server. Kubectl can be used from any client machine which can connect to the cluster without any special setup.

#### Distributed Key-Value Store (etcd)

Etcd is a distributed key-value store and provides a reliable way for sharing data within a cluster. It is the key component for the communication between the Kubernetes Master and the Kubernetes Worker-Nodes. The Kubernetes Master provides configuration for the Kubernetes Nodes and retrieves state information from the Kubernetes Worker-Nodes [Cor18].

#### Kubernetes API-Server (kube-apiserver)

The Kubernetes API-Server exposes the interface for interacting with the Kubernetes Cluster and is located on the Kubernetes Master. It represents the frontend of the Kubernetes Cluster and provides all necessary API to manage the cluster and the Pods running on it.

#### Kubernetes Scheduler (kube-scheduler)

The Kubernetes Scheduler watches the Kubernetes Cluster for newly created Pods and assigns the Pods to a Kubernetes Worker-Node. The Kubernetes Scheduler decides which Kubernetes Worker-Node is suitable for the Pod. Multiple factors are taken into account for scheduling decisions such as individual specifications, resource requirements, available resources and hardware/policy/software constraints.

#### Kubernetes Controller Manager (kube-controller-manager)

The Kubernetes Controller Manager is responsible for the managing of the different controllers. A Kubernetes Controller is running in a loop and ensures that the state of the system is valid, depending on the controller type. For instance, the replication controller ensures the correct number of Pods for each replication controller object

within the Kubernetes Cluster. Kubernetes provides a set of controllers such as a replication controller, node controller, endpoint controller and service account controller.

### 4.2.3 Kubernetes Worker

The Kubernetes Worker is a node within the Kubernetes Cluster which acts as the slave node which hosts the Pods and is managed by the Kubernetes Master. The Kubernetes Worker can be a VM or a physical machine depending on the Kubernetes Cluster setup. It contains the Kubernetes Runtime-Environment and Docker. The following sections briefly introduce the Kubernetes Worker-Components, which are responsible for running the Pods on the Kubernetes Worker-Node [Kub18b].

#### Kubernetes Agent (kubelet)

The Kubernetes Agent is a process running on the Kubernetes Worker-Nodes which interacts with the Kubernetes Master via the Kubernetes API-Server. The Kubernetes Agent ensures that the containers are running in a Pod as specified by the provided Pod specifications. The Pod specifications can be provided by an file in a specific directory (gets periodically checked), or via the Kubernetes API-Server.

#### Kubernetes Network-Proxy (kube-proxy)

The Kubernetes Network-Proxy manages the networks defined by the specifications and reflects the services which are bound to a Pod. It can perform simple TCP and UDP forwarding and can be connected to multiple backends. Any communication of a Pod to another Pod or to the Internet is handled by the Kubernetes Network-Proxy.

#### Container Runtime

The container runtime is the software responsible for running the containers on the Kubernetes Worker. Kubernetes supports multiple container runtimes, but usually its Docker which has been discussed in Chapter 3 on page 8.

Kubernetes provides all features to implement a dynamic scalable service infrastructure such as workflows for rolling out services, replica management, secret and configuration management, which enterprises can profit from. Secrets are protected from being accessed by the developer and configurations can be applied without building a new service artifact. Kubernetes enhances Docker with orchestration tooling necessary to run large scale dynamic containerized service infrastructures. Nevertheless, sometimes even Kubernetes is not suitable enough for some use cases, which can be overcome with PaaS platforms like Openshift, which is discussed in the following Chapter **??** on page ??.

# References

## Literature

[Cor14]     Intel Corporation. *Linux\* Containers Streamline Virtualization and Complement Hypervisor-Based Virtual Machines.* Intel Corporation, 2014 (cit. on pp. 13, 14).

[DG15]      T. Devi and R. Ganesan. *Platform-as-a-Service (PaaS): Model and Security Issues.* School of Computing Science and Engineering, VIT University, 2015 (cit. on p. 1).

[HW08]      Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns.* 11th ed. Boston, MA: Pearson Education, Inc., 2008 (cit. on p. 1).

[Kie16]     Kief Morris. *Infrastructure as Code: Managing Servers in the Cloud.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2016 (cit. on pp. 3–6).

[Liu+15]    Lawrence Liu et al. *Integration Platform as a Service: The next generation of ESB, Part 1.* IBM, 2015 (cit. on p. 1).

[Sch14]     Mathijs Jeroen Scheepers. *Virtualization and Containerization of Application Infrastructure: A Comparison.* University of Twente, 2014 (cit. on p. 8).

## Online sources

[Ama18]     Amazon Web Services (AWS). *Amazon EKS.* 2018. URL: https://aws.amazon.com/eks/ (visited on 03/31/2018) (cit. on p. 15).

[Clo18]     Cloud Native Computing Foundation. *Sustaining and Integrating Open Source Technologies.* 2018. URL: https://www.cncf.io/ (visited on 03/31/2018) (cit. on p. 15).

[Cor18]     CoreOS. *etcd.* 2018. URL: https://coreos.com/etcd/docs/latest/ (visited on 04/01/2018) (cit. on p. 18).

[Cos12]     Costa, Glauber. *Resource Isolation: The Failure of Operating Systems and How We Can Fix It.* 2012. URL: https://linuxconeurope2012.sched.com/event/bf1a2818e908e3a534164b52d5b85bf1 (visited on 03/30/2018) (cit. on p. 12).

[Doc18a]    Docker Inc. *Docker Machine Overview*. 2018. URL: https://docs.docker.c
            om/machine/overview/ (visited on 03/17/2018) (cit. on p. 11).

[Doc18b]    Docker Inc. *Dockerfile Reference*. 2018. URL: https://docs.docker.com/e
            ngine/reference/builder/ (visited on 03/09/2018) (cit. on p. 9).

[Doc18c]    Docker Inc. *Dockerfile Registry*. 2018. URL: https://hub.docker.com/
            (visited on 03/09/2018) (cit. on p. 9).

[Doc18d]    Docker Inc. *Overview of Docker Compose*. 2018. URL: https://docs.dock
            er.com/compose/overview/ (visited on 02/23/2018) (cit. on p. 6).

[Doc18e]    Docker Inc. *Swarm mode overview*. 2018. URL: https://docs.docker.com
            /engine/swarm/ (visited on 03/30/2018) (cit. on p. 14).

[Doc18f]    Docker Inc. *What is Docker*. 2018. URL: https://www.docker.com/what
            -docker (visited on 02/23/2018) (cit. on p. 8).

[Goo18]     Google Cloud. *Kubernetes Engine*. 2018. URL: https://cloud.google.com
            /kubernetes-engine/ (visited on 03/31/2018) (cit. on p. 15).

[Heo15]     Heo, Tejun. *CGROUPS*. 2015. URL: https://www.kernel.org/doc/Docum
            entation/cgroup-v2.txt (visited on 03/30/2018) (cit. on p. 13).

[Kub18a]    Kubernetes. *API Overview*. 2018. URL: https://kubernetes.io/docs/refe
            rence/generated/kubernetes-api/v1.10/ (visited on 04/01/2018) (cit. on
            p. 17).

[Kub18b]    Kubernetes. *Kubernetes Components*. 2018. URL: https://kubernetes.io
            /docs/concepts/overview/components/ (visited on 04/01/2018) (cit. on
            pp. 18, 19).

[Kub18c]    Kubernetes. *Pods*. 2018. URL: https://kubernetes.io/docs/concepts/workl
            oads/pods/pod/ (visited on 04/01/2018) (cit. on p. 17).

[Kub18d]    Kubernetes. *Production-Grade Container Orchestration*. 2018. URL: htt
            ps://kubernetes.io/ (visited on 03/30/2018) (cit. on pp. 14, 15).

[Kub18e]    Kubernetes. *Services*. 2018. URL: https://kubernetes.io/docs/concepts/se
            rvices-networking/service/ (visited on 04/01/2018) (cit. on p. 17).

[Lin18a]    Linux Containers. *Linux Containers*. 2018. URL: https://linuxcontainers
            .org/ (visited on 03/29/2018) (cit. on p. 8).

[Lin18b]    Linux Foundation. *namespaces (7)*. 2018. URL: http://man7.org/linux
            /man-pages/man7/namespaces.7.html (visited on 03/30/2018) (cit. on
            p. 14).

[Men18]     Menage, Paul and Lameter,Christoph. *CGROUPS*. 2018. URL: https://w
            ww.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (visited on
            03/30/2018) (cit. on p. 13).

[Mic18]     Microsoft Azure. *Azure Container Service (AKS)*. 2018. URL: https://d
            ocs.microsoft.com/en-us/azure/aks/ (visited on 03/31/2018) (cit. on
            p. 15).

[Red17]     Red Hat Inc. *What is Wildfly?* 2017. URL: http://wildfly.org/about/
            (visited on 02/23/2018) (cit. on p. 6).

[Red18a]    Red Hat Inc. *Red Hat JBoss Fuse*. 2018. URL: https://developers.redhat
            .com/products/fuse/overview/ (visited on 02/22/2018) (cit. on p. 1).

[Red18b]    Red Hat Inc. *Red Hat Openshift Container Platform*. 2018. URL: http
            s://www.openshift.com/container-platform/features.html (visited on
            02/22/2018) (cit. on p. 2).