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. 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.

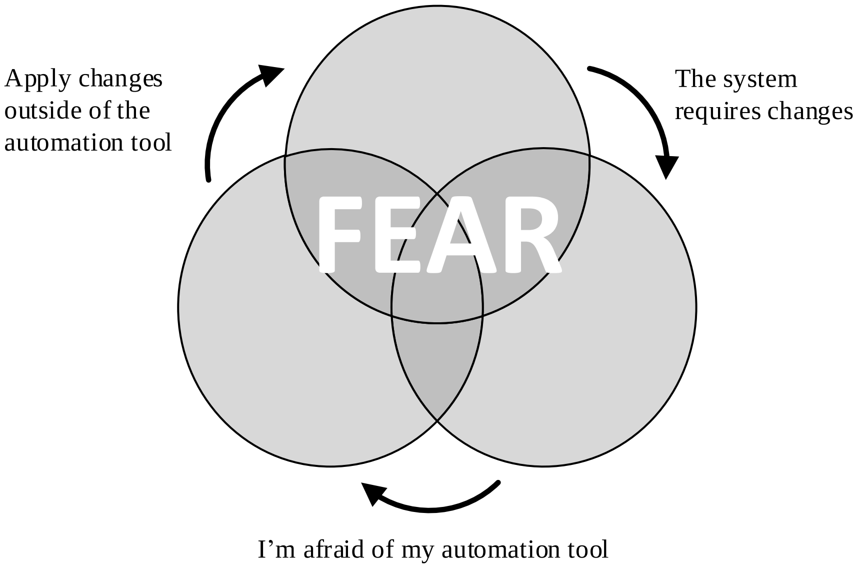
Apply changes outside of the automation tool
The system requires changes
FEAR
I'm afraid of my automation tool

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. 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.

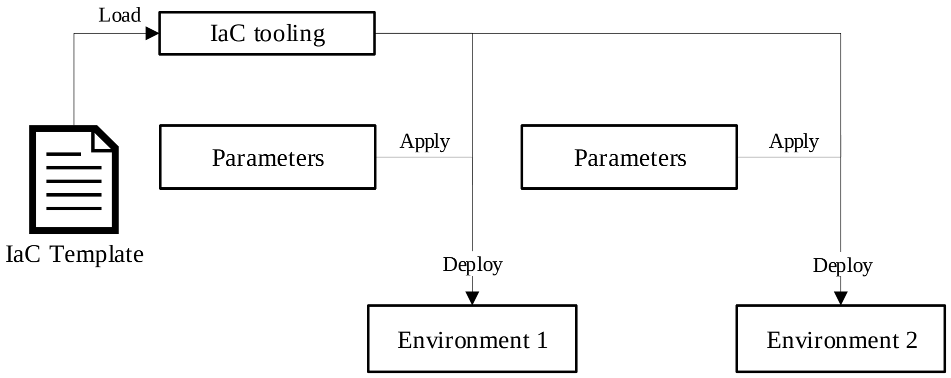

Figure 2.2: Schema of a parametrized infrastructure deployment

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.

```yaml
version: "2.1"
services:
   wildfly:
        container_name: wildfly
        image: wildfly:latest
        ports:
            - "${EXPOSED_PORT}:8080"
        environment:
            - "POSTGRES_DB_URL=${POSTGRES_DB_URL}"
            - "POSTGRES_DB_NAME=${POSTGRES_DB_NAME}"
            - "POSTGRES_USER=${POSTGRES_USER}"
            - "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) [Doc18e; Lin18]. 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.

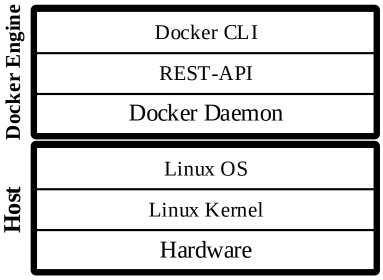

Figure 3.1: Docker Engine architecture

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 Damon

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 processes, 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 architecture. 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.

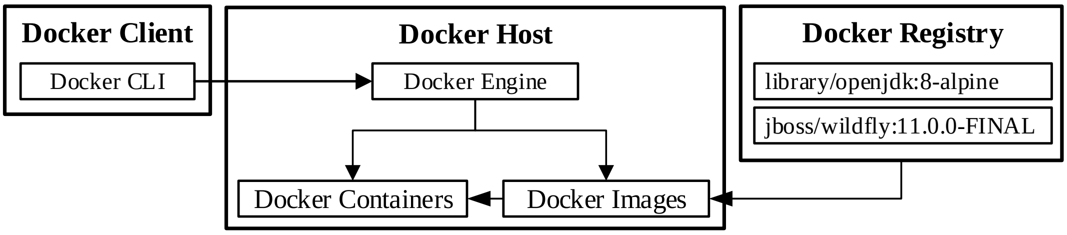

Figure 3.2: Docker Architecture

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.

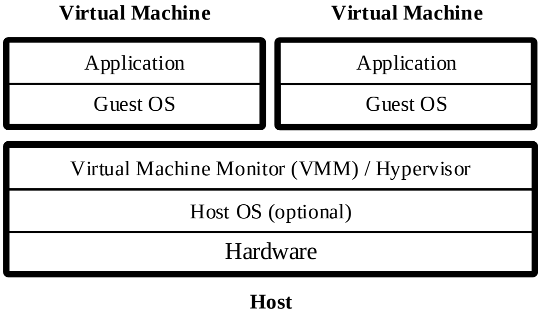

Figure 3.3: Architecture of virtualized applications

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.

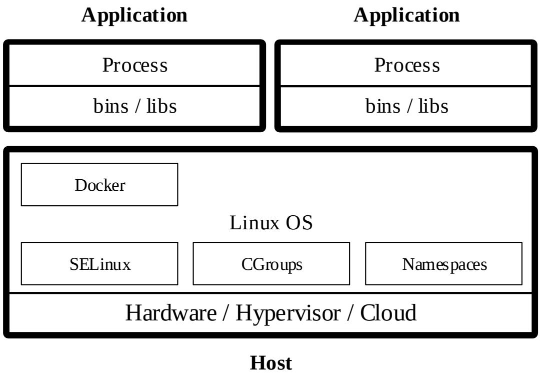

Figure 3.4: Architecture of containerized applications

The two main technologies underlying LXC are *CGroups* and *Namespaces*. These two kernel technologies provide the features 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 group and namespaces manage the view of the system to a group. A container is managed in a namespace and therefore has a limited view of the system such as networks or other processes. Namespaces provide the isolation of a container.

References

Literature

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

[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

[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.com/machine/overview/ (visited on 03/17/2018) (cit. on p. 11).

[Doc18b] Docker Inc. *Dockerfile Reference.* 2018. URL: https://docs.docker.com/engine/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.docker.com/compose/overview/ (visited on 02/23/2018) (cit. on p. 6).

[Doc18e] Docker Inc. *What is Docker*. 2018. URL: https://www.docker.com/what-docker (visited on 02/23/2018) (cit. on p. 8).

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

[Lin18] Linux Containers. *Linux Containers*. 2018. URL: https://linuxcontainers.org/ (visited on 03/29/2018) (cit. on p. 8).

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

[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: https://www.openshift.com/container-platform/features.html (visited on 02/22/2018) (cit. on p. 2).