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 [Red18c]. 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+15a].

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 [Red18d]. 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.

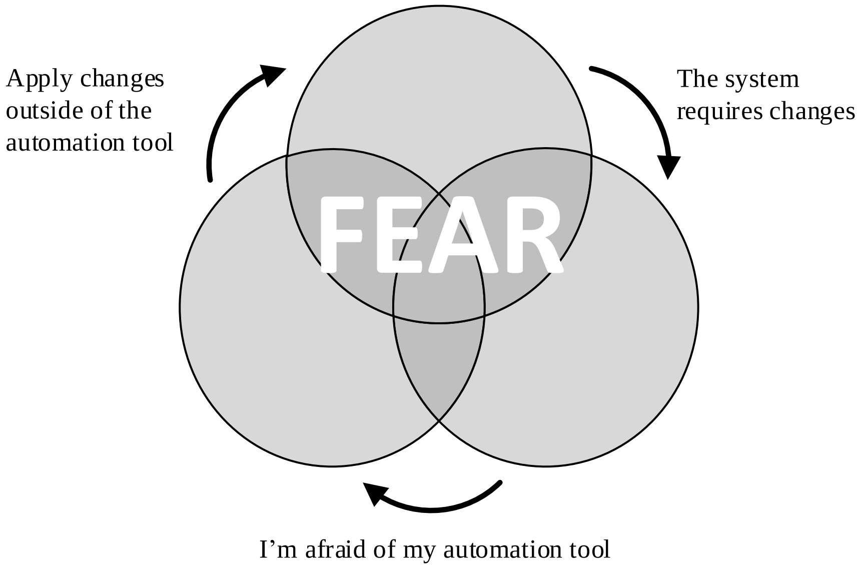

Figure 2.1: Automation Fear Spiral

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.

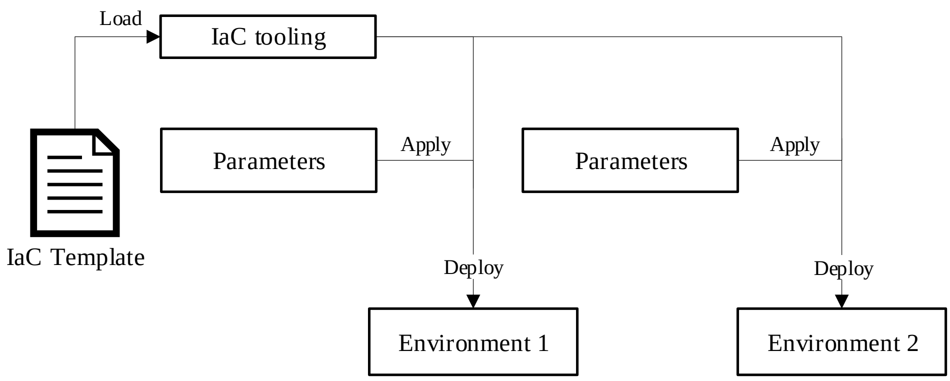

Figure 2.2: Schema of a parametrized infrastructure deployment

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.

```
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) [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.

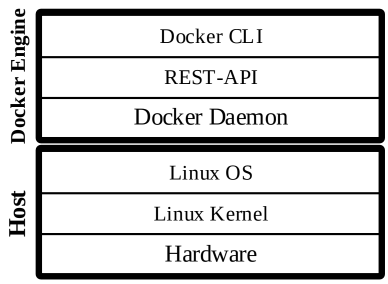

Figure 3.1: Docker Engine architecture

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 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.

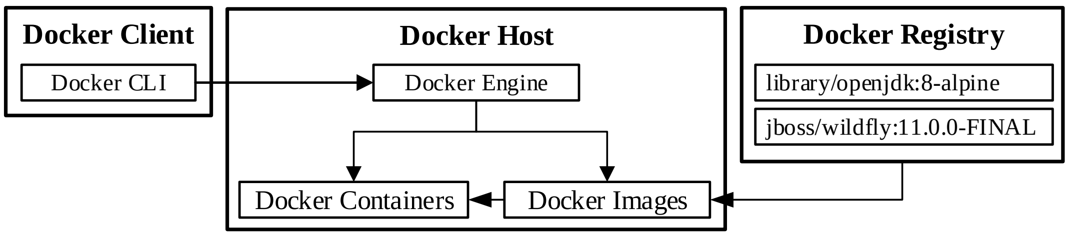

Figure 3.2: Docker Architecture

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.

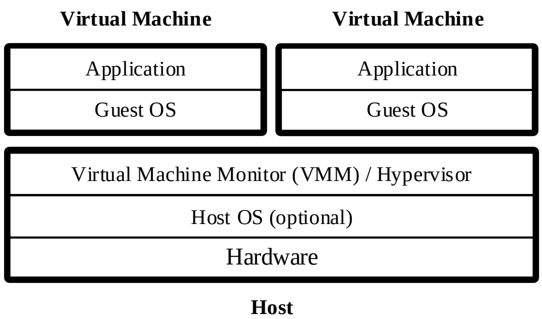

Figure 3.3: Architecture of virtualized applications

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.

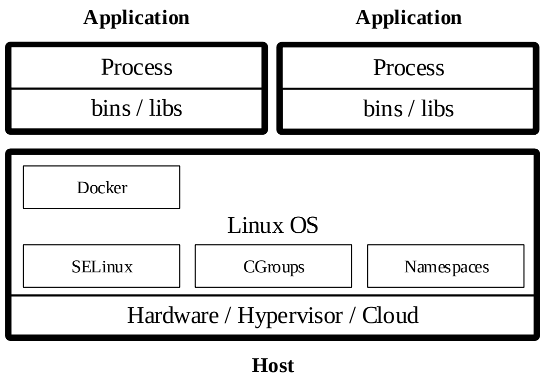

Figure 3.4: Architecture of containerized applications

The two 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 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 [Ama18a; Goo18b; Kub18d; Mic18b].

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 [Clo18a].

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.

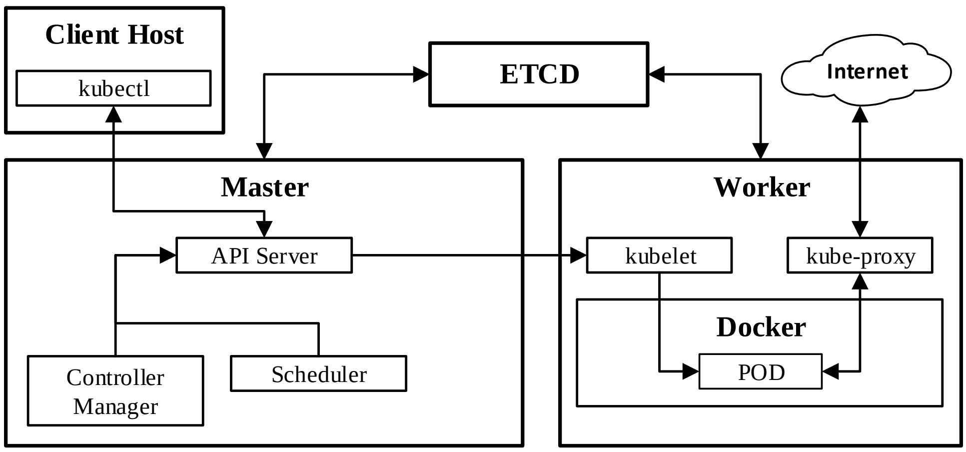

Figure 4.1: Architecture of a Kubernetes Cluster

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 5 on the next page.

Chapter 5

Platform as a Service with Openshift

Platform as a Service (PaaS) is a cloud service which provides an on demand platform for building, deploying and running containerized applications in the cloud. PaaS can be seen as an enhancement of CaaS, which has been discussed in the former Chapter 4 on page 15. A PaaS platform does not only provide a container runtime for running containers in the cloud but also tooling for building, deploying and monitoring of containerized applications as well as security mechanisms for securing those applications. There are multiple PaaS providers on the market but the most popular PaaS providers are RedHat Openshift Online, Microsoft Azure Cloud Services, Google App Engine and AWS Elastic Beanstalk. They all bring in their own flavor of PaaS but they all provide similar features necessary by an PaaS platform [Ama18b; Goo18a; Mic18a; Ope18a].

PaaS providers usually provide templates for the major programming languages and application servers, and integration to other cloud services as well. External cloud services of the same vendor are usually better supported than cloud services of other vendors. This is normal, because cloud providers want the developers to use their service over the services of the competition. What all PaaS providers have in common is the consumption based pricing model, where only the consumed physical resources have to be paid for.

IPaaS can be seen as an enhancement of PaaS which is suitable for implementing an ESB which is discussed in Chapter 6 on page 26. IPaaS enhances an ordinary PaaS platform by providing tooling for integrating external service effortlessly, via a low/no code platform, where services can be integrated via an UI, rather then by implementing source code. RedHat JBoss Fuse 7 is an example for an IPaaS platform which will replace JBoss Fuse 6.x in the near future [Liu+15a; Liu+15b; Red18a].

Openshift Origin is an open source PaaS platform, which has been released in April 2012 and is the upstream project for Openshift. Before Openshift 3 (Jun 2013), Openshift used its own container runtime and orchestration tooling, which since Openshift 3 have been replaced by Docker and Kubernetes, because of its popularity and general availability. Openshift is the only major PaaS platform of the formerly noted ones which can be self hosted or hosted by a local provider. The other formerly noted PaaS providers such as Microsoft Azure are only available as a cloud service hosted in the vendors data centers [Ope18b].

5.1 The need for Platform as a Service

As mentioned in Section 4.2.3 on page 19, there are some use cases where Kubernetes or in particular CaaS is not suitable anymore. CaaS is suitable if its used by developers, but not for persons without any deep knowledge of Docker and Kubernetes. This is where PaaS platforms come into place, which provide a web console and a template mechanism, which can be used by non-developers. Developers specify templates for the provided services which contains all technical parts of a service infrastructure and non-developers provide values for the exposed parameters which are non-technical, and the PaaS platform instantiates the template and deploys the service infrastructure automatically.

Enterprises can profit from PaaS platforms by defining templates for services they provide for their departments, partners or customers, who can create an instance of a provided service on demand, and destroy it if not needed anymore. PaaS platforms provide a self service console, where services can be created, managed and destroyed effortlessly without the need to understand the underlying technology. The self service console could be implemented by enterprises for their specific use cases, where the self service console interacts with PaaS platform via its exposed API.

PaaS platforms like Openshift usually provide an integration in a Continuous Integration / Continuous Deployment (CI/CD) workflow, which allows to automatically build and deploy new service releases in the PaaS platform automatically via web hooks. Therefore, the PaaS platforms are integrated in the whole software life cycle. This decreases the effort of the developers to interact with the cloud platform and provide additional automation.

5.2 Openshift

Openshift is a open source PaaS platform, which uses Docker and Kubernetes for the Docker Container orchestration. Openshift is designed as a client server architecture and a master slave architecture, the same way as a Kubernetes Cluster, which has been discussed in Section 4.2 on page 16. An Openshift Cluster can contain multiple Kubernetes Clusters which are managed by a Openshift Master-Node, which is discussed in Section 5.2.1 on the following page, which manages the Kubernetes Master-Nodes. Openshift provides Openshift Projects, which are discussed in Section 5.2.2 on the next page, which place all defined resources in a Kubernetes namespace, and which are isolated form each other. The following Figure 5.1 on the following page illustrates the architecture of an Openshift Cluster [Ope14; Ope18c].

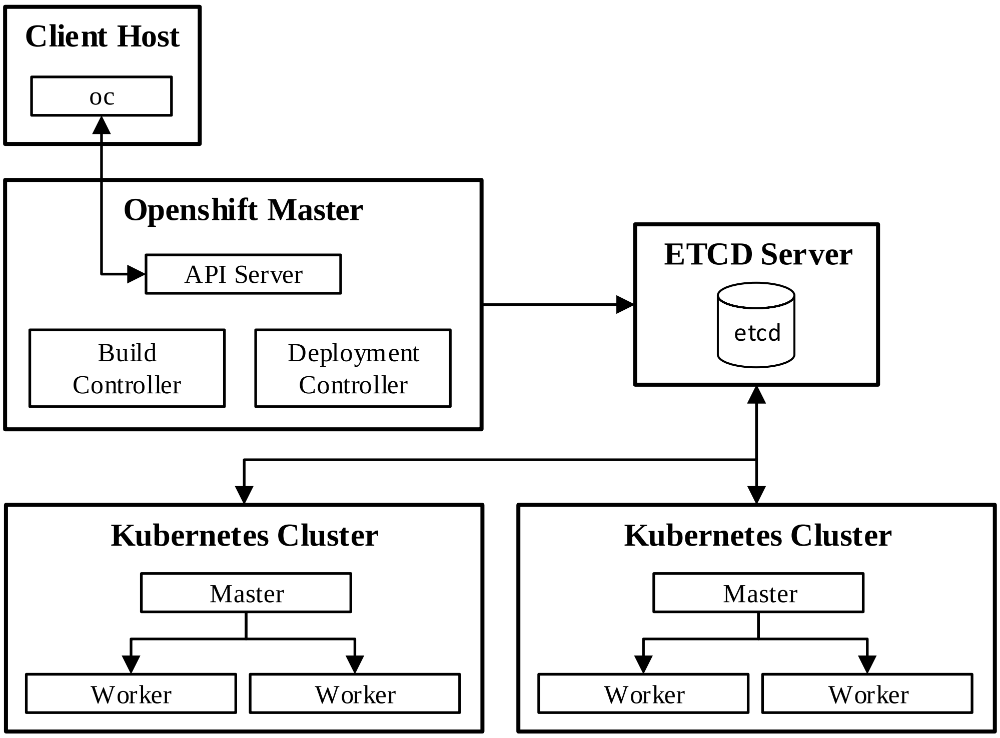

Figure 5.1: Architecture of a Openshift Cluster

5.2.1 Openshift Master

The Openshift Maser-Node manages the Kubernetes Master-Nodes of the Kubernetes Clusters the Openshift Cluster contains. The Openshift Master exposes a REST-API via the clients can interact with the Openshift Cluster. Therefore that Openshift is placed on top of Kubernetes, the Openshift Master-Node acts similar as a Kubernetes-Master-Node, which has been discussed in Section 4.2.2 on page 18. Additionally Openshift provides features Kubernetes does not, such as a role and group based security model for isolating the Kubernetes Namespaces via Openshift Projects and controllers for managing the additional Openshift Objects. The following Section 5.2.2 discusses Openshift Projects, which are the main feature provided by Openshift.

5.2.2 Openshift Project

An Openshift Project represents a Kubernetes namespace, where all resources of an Openshift Project are located. An Openshift Project provides the isolation and security Kubernetes Namespaces do not provide. The Figure 5.2 on the next page illustrates the Openshift Project-Architecture, its contained Objects and their dependency to each other. The bold marked objects within the Project representing the Openshift Objects which are provided by Openshift.

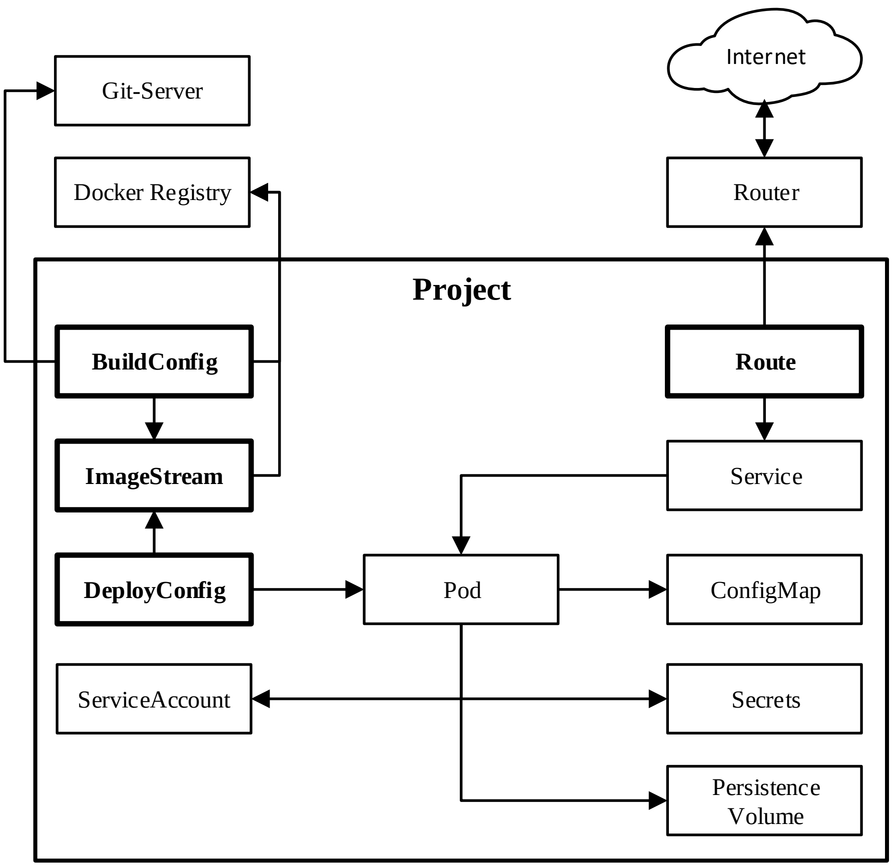

Figure 5.2: Architecture of a Openshift Project

Openshift Objects are persistent objects in the Openshift System, and the Openshift Objects describe the state of the Openshift Cluster. This behavior has been inherited from the underlying Kubernetes System as discussed in Section 4.2.1 on page 16. The following sections briefly introduce the new Objects provided by Openshift.

BuildConfig

A Build Configuration specifies the way how a Docker Image is built on the Openshift platform. The built Docker Image is pushed into the Openshift internal Docker Registry. Openshift Build Configurations support the following listed strategies:

- The *Source-to-Image (S2I)* strategy is the build strategy which builds a Docker Image from source code.
- The *Docker* strategy is the build strategy which builds a Docker Image from a Dockerfile.

- The *Custom* strategy is the build strategy which build a Docker Image with a custom implemented build mechanism.
- The *Pipeline* strategy is the build strategy which performs a Jenkins pipeline build on a Jenkins build server.

The necessary resources for the particular build strategy are provided via a git repository, and a Build Configuration can be triggered by an external service such as Github via a web hook [Ope18d].

ImageStream

An Image Stream and its Image Stream-Tags are an abstraction of the actual used Docker Image and an Image Stream uses the same naming convention as Docker Tags (E.g *myproject/app:1.0*), where

- *myproject* represents the Image Stream namespace,
- *app* represents the Image Stream name and
- *1.0* represents the Image Stream-Tag.

An Image Stream-Tag references the actual Docker Image by its tag. Once the Docker Image has been imported, it will not be automatically pulled again unless the Image Stream-Tag has the name *latest* which causes Openshift to always to pull the referenced Docker Image.

A Docker Image can be updated in a Docker Registry, which would break the consistency principle, because it wouldn't be the same Docker Image as used before the update. An Image Stream or in particular the Image stream-Tag prevents this, by referencing the actual Docker Image instance instead of only referencing the Docker Image by its tag. This approach makes the Docker Image immutable within a Openshift Project, unless the latest version is explicitly defined.

DeployConfig

A Deployment Configuration specifies how a deployment of an Pod has to be performed. A Deployment Configuration allows to specify the Kubernetes life cycle hooks pre-hook or post-hook, which are used to configure the deployed Pod before its process has started (pre-hook) or after its process has started and is ready (post-hook). Deployment Configurations support the following listed deployment strategies:

- The *Rolling* strategy is the deployment strategy which waits for the new deployment to be ready before the old deployment gets removed.
- The *Recreate* strategy is the deployment strategy which removes the old deployment when the new deployment gets started.
- The *Custom* strategy is the deployment strategy which performs the deployment by a custom implementation.

Route

A Route exposes a Service with a host name to an external network (mostly the Internet), so that it can be reached by its host name from clients located outside of the Openshift Cluster. The Route is deployed on a Openshift Router, which performs the routing between the external network and the connected Service. A Route can be secured with TLS, where the certificates of the Openshift Cluster can be used or the certificate can directly be defined in the Route definition.

Chapter 6

Enterprise Service Bus

An Enterprise Service Bus (ESB) is a architectural pattern which describes a distributed computing architecture, where distributed services are interacting with each other via the ESB. The ESB pattern if part of the Service Oriented Architecture Patterns (SOA).

An ESB in the industry is mostly be taken as a third party middleware, which provides features for implementing integrations with the Enterprise Integration Patterns (EI). Enterprises use third party middleware like JBoss Fuse for implementing integration services, which integrate external and internal services. JBoss Fuse is based on the Enterprise Application Platform (EAP), and bundles common frameworks used for integrations such as Camel, and is responsible for orchestration and mediation of the services.

The integration of internal as well as external services has become more important over time, especially with the upcoming of cloud solutions which provide PaaS. Common ESB middleware on the market usually define the ESB as a single application, which contains all integration services. But the need for flexibility and shorter response times drives enterprises to split up their teams and integrations. This leads to separated services, which are managed as microservices, which have their own life cycle. De-coupling of teams leads to de-coupling of services, where the services need to provide a well defined and managed public API [HW08; Red05; The15].

6.1 The need for an Enterprise Service Bus

Enterprises need an ESB to provide integrations between internal and external services or both, where the integration is meant to provide a business value for the enterprise. An integration of an external service could enhance the reach of a customer, who now could be able to consume external partner services via the enterprise provided infrastructure or product. In the digital age, it is normal to consume a digital service like Netflix, which runs a streaming service. Thus, integrations an enterprise has to provide and maintain will become more over time.

6.2 Architecture

Figure 6.1 illustrates the conceptional architecture of an ESB, which orchestrates and mediates integration services. A service can act either a producer service, which gets accessed by clients, or can act as a consumer service, which acts as the client for a producer service. The ESB is responsible for orchestration, mediation, security, transformation, routing and service discovery, whereby these aspects are covered by a ESB middleware provided frameworks and libraries. Additionally, an ESB middleware provides libraries which help to implement Service Components under consideration of the SOA and EI Patterns [HLA05; Mas18].

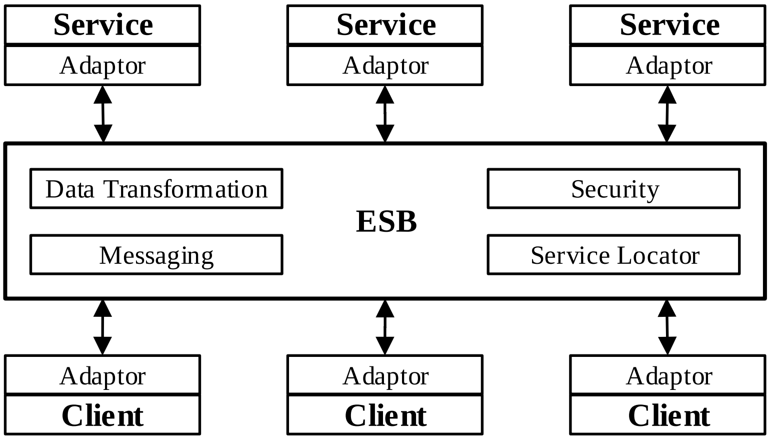

Figure 6.1: Architecture of an ESB

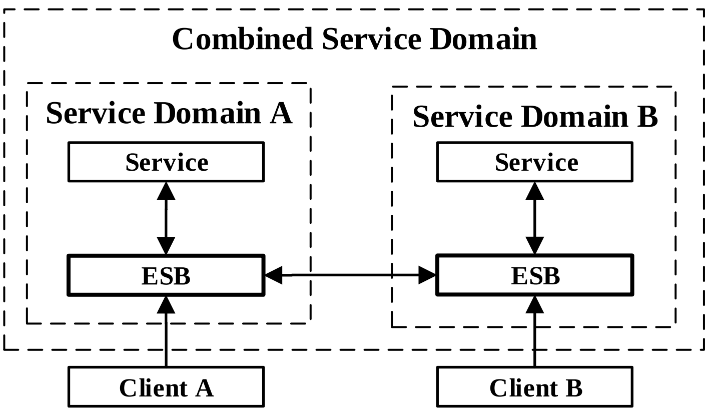

Figure 6.2: Architecture of a bi-directional enterprise integration

Figure 6.2 on the preceding page illustrates a bi-directional integration of services between two partner enterprises, where each integrated service is allowed to be consumed by the partner's customers, but only if the services is accessed via the partner's infrastructure. The ESB of the enterprises integrate the partner's provided service into their service domain, which can be accessed by their customers. For instance, an IP-TV provider can be integrated by an Internet Service-Provider (ISP), to provide Internet TV to their customers.

6.3 ESB with EAP

An ESB is an architectural pattern for a distributed system, and has been implemented in software to provide an integration platform to developers, so that they can implemented integration services. Before the upcoming of the cloud, ESB implementations used existing platforms such as EAP, OSGI or Karaf for the service orchestration and mediation. In case of EAP, the platform provides all libraries and frameworks developers need for implementing an integration service. Mostly, the integration services are managed within a single application, which represents the ESB. This is a monolithic approach of organizing integration services, but has the advantage that the management of the services is easier, because the source code is not separated, and therefore the implementations of all services needs to be consistent at compile time. Figure 6.3 illustrates the monolithic organization of the integration services within a single ESB application, which is hosted on EAP.

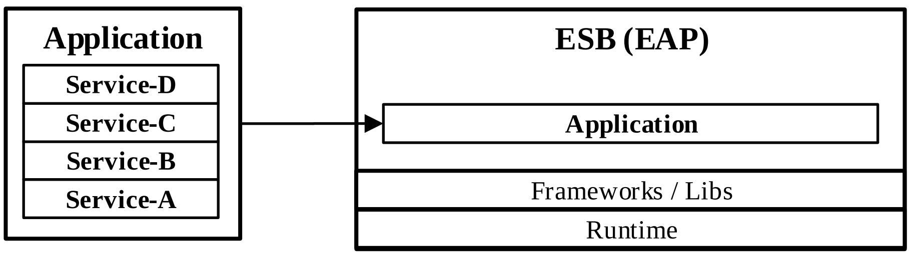

Figure 6.3: ESB running on EAP

With the upcoming of cloud platforms such as PaaS, the cloud can now take over the mediation, orchestration and security aspects of an ESB. The integration services can be completely separated and de-coupled from each other, designed as microservices with their life cycle which can be managed by the cloud platform. Additionally, the integration service are hosted in a clustered environment, which allows them to be distributed among multiple nodes, which increases fail over security. The new term for this kind of ESB is IPaaS, whereby the ESB is represented by an PaaS platform such as Openshift, which provides all tooling for implementing integration services [Liu+15a; Liu+15b].

6.4 ESB with Openshift

With the upcoming and general availability of cloud platforms like PaaS, it was possible to move an ESB into the cloud, whereby the cloud platform takes over some aspects of the ESB middleware such as mediation and orchestration. A main problem of existing ESB implementations is the fact, that all integration services are managed within a single application, which represents the ESB. If the ESB is an cloud platform, the integration services have to be implemented as microservices, which forces developers to separate their integration services into separate code bases and to provide a proper designed and managed public API for their services. Figure 6.4 illustrates the integration services of an ESB application, which is running on Openshift.

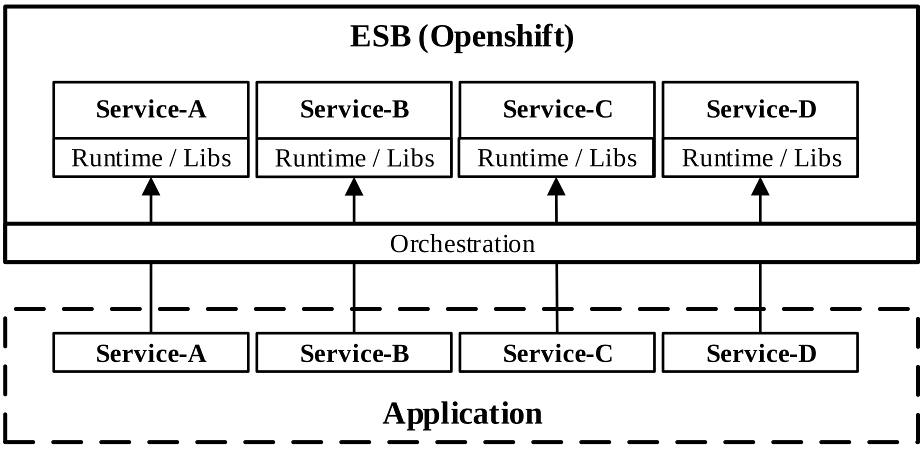

Figure 6.4: ESB application architecture with Microservices

As discussed in the introduction of this Chapter, enterprises need to separate their integrations and teams, to be faster and more responsive to business changes. Therefore, the microservice architecture, which is necessary when the ESB is represented by a cloud platform, can help enterprises to separate their teams and integration services.

6.5 Integration example

This Section will discuss a integration and how it would have been designed as part of a monolithic ESB application. The integration discussed in this chapter is the base for the prototype of this thesis, which is specified in Chapter 7 on page 32. Figure 6.5 on the following page illustrates the integration, contained services and involved service domains. The integration service integrates an external database with an internal application, which is consumed by a public client. How the database is allowed to be accessed, is implemented in the integration service, which is the only service allowed to communicate with the external located database.

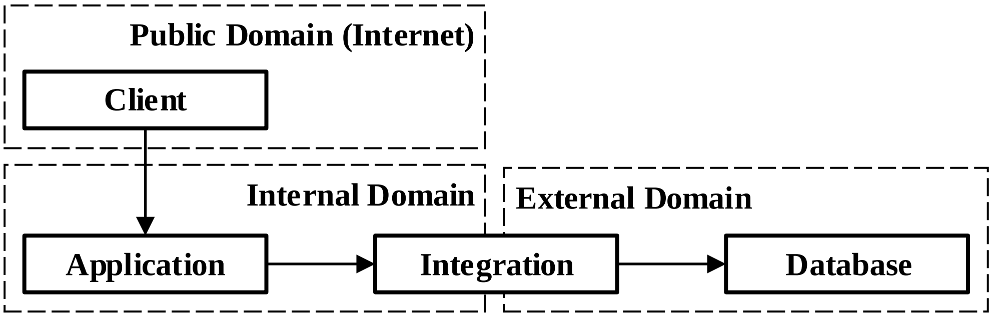

Figure 6.5: Integration Service-Domains

Figure 6.6 illustrates the design of the integration in an monolithic ESB application with the use of Service Component Architecture (SCA). A service within the ESB application is represented by a Service Component, which exposes consumable services *(Service)* and is connected to other services or external systems *(Reference)*.

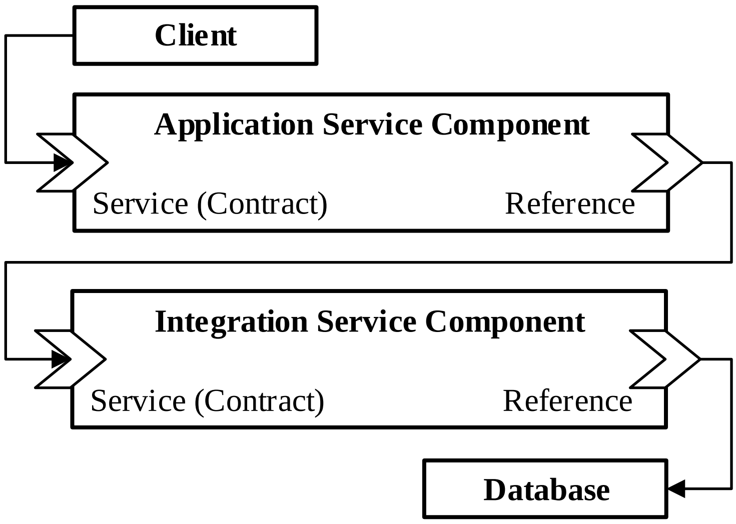

Figure 6.6: Integration Services with SCA

ESB middleware, such as JBoss Fuse, provides frameworks and libraries, which implement the SCA patterns and provide a lightweight way of implementing Service Components. The Service Component orchestration, mediation is done by the ESB middleware. Additionally, the ESB middleware provides bindings for commonly used technologies such as REST or SAOP, for Services and References. Thus, the developers don't have to setup for instance a REST-Server or REST-Client anymore, but only need to provide the service contract and connection settings [Lop08; Ric15].

The next Chapter 7 on the following page will specify the prototype, which will implement the integration, which was discussed in this Section. The integration will be implemented with the microservice architecture, whereby Openshift will act as the ESB, where the services are running on. The service structure will be the same as illustrated in Figure 6.6 on the previous page, whereby the Service Components will become microservices, which bring in their own runtime environment and libraries. Openshift will take over the orchestration, mediation and security aspects of an ESB middleware.

Chapter 7

Design ESB in Openshift

In this chapter, the ESB integration as discussed in Section 6.5 on page 29, will be designed as microservices, which can run on a Openshift Cluster. A Openshift Project will act as the ESB, which will provide the orchestration, mediation, configurations and secrets for the integration services. The concrete functionality of the services is considered not to be important. The goal is to redesign the Service Components of Figure 6.6 on page 30 as microservices, and to design the Openshift Project, which will host the integration services.

As discussed in Chapter 6 on page 26, an ESB is a distributed computing architecture, where distributed services act as a consumer or producer. These integration services provide a business value for an enterprise in form of an integration of an internal or external service. There are multiple providers of ESB middleware like JBoss Fuse, which provide tooling for implementing Service Components running on an ESB. The prototype will illustrate that SOA Service Components can be implemented as microservices, where features provided by the ESB middleware such as mediation and bindings will have to be replaced by other implementations.

7.1 Microservice Architecture

Figure 7.1 on the next page illustrates the microservice architecture of the integration prototype, based on the architecture of Figure 6.6 on page 30. Conceptually, the transformation of a Service Component to a microservice is easy, because a Service Component and a microservice act very similar. Compared to a Service Component of a ESB middleware, a microservice is completely separated from the other services, brings in its own dependencies and runs in its own runtime environment. From the implementation point of view, the microservice needs to provide an runtime environment, which formerly was provided by the ESB middleware. Therefore, that the microservices are completely separated, the access of other services is not mediated as usual anymore, because the communication is now performed via standard protocols like HTTP(S). In a monolithic ESB application running on EAP, every Service Component accessed via a Reference uses the actual service instance within the same runtime environment. With microservices, running in their own runtime environment, it is not possible to access the service instance directly anymore. Only communication via standard protocols such as HTTP(S) is supported.

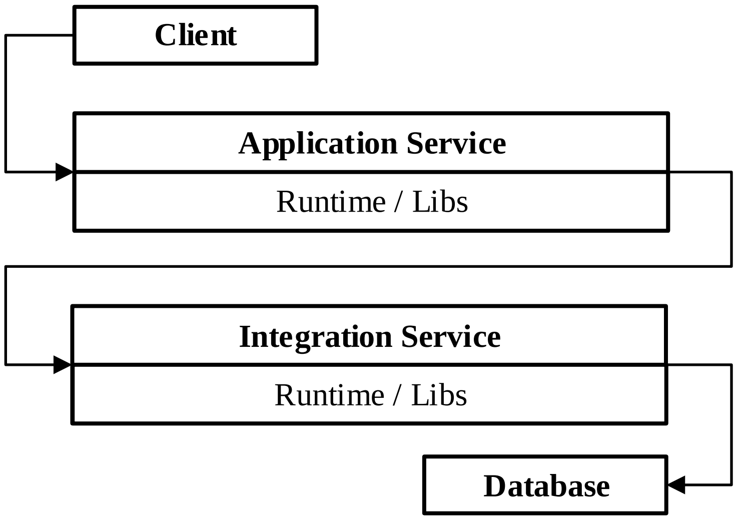

Figure 7.1: Microservice Architecture

7.2 Microservice Aspects

In this section the necessary aspects of microservices are discussed, which will ensure that the integration services are properly implemented and can effortlessly be managed. Especially the monitoring of the integration services becomes very important when moving from a conventional monolithic ESB application, like an application running on JBoss Fuse, to microservice based integration services. Integration services running as microservices on Openshift, are running in their own runtime environment and therefore cannot share any runtime resources, which are available in a monolithic application.

Figure 7.2 on the next page illustrates the hierarchy of the microservice aspects, which are discussed in the following sections. The in the following discussed microservice aspects shall ensure that the microservices are

- secure,
- configurable for multiple environments,
- observable by developers and operators,
- resilient to failures
- and measurable.

Microservice Aspects
API Management
Fault Tolerance
Metrics
Logging
Tracing
Configuration
Security

Figure 7.2: Service Requirements

7.2.1 Security

Distributed microservices need to be secured properly from unauthorized access, therefore the microservices will be secured with OAuth. OAuth is a token based authentication scheme, which has become popular over past few years. There are several open source implementations for interacting with a authentication service via the OAuth scheme. The Integration Service must authenticate its client the Application Service against an central authentication service, whereby the Application Service will retrieve the access tokens from the central authentication service [oau18].

7.2.2 Configuration

The MicroProfile specifications provide the MicroProfile-Config specification, which provides an API to inject configuration parameters into objects. The injected configuration parameters are loaded from so called configuration sources. A configuration source can either be Java System-Properties, Environment Variables, Properties Files, YAML Files or custom implementations for instance to retrieve configurations from a database. The microservices must be implemented in a way to be configurable for different stages such as DEV (development), TEST (testing) and PROD (productive environment), whereby the services are only allowed to use configuration parameters via injection [Ecl18a].

7.2.3 Tracing

The MicroProfile specifications provides the OpenTracing specification, which provides an API for tracing an application on a method level or across service boundaries. Distributed Tracing allows to comprehend service or method call chains of distributed services. There is open source tooling available to analyze the collected tracing data provided by the distributed services. The services must collect reasonable tracing information and send this data to a central tracing service [Clo18b].

7.2.4 Logging

Distributed Logging allows to comprehend logs across service boundaries within a service call chain, where the logs of all involved services have to be marked with the same transaction id. There is open source tooling available to analyze the collected logs. The services must provide all of their logging to a central service, whereby the logs are marked with a transaction id, which is provided by the OpenTracing API. Optionally, the services are allowed to add additional markers, which can help developers and operators to analyze problems or to group service logs.

7.2.5 Metrics

The MicroProfile specifications provide the specification MicroProfile-Metrics, which provides an API to define and manage metrics. Metrics allow to comprehend the state of a microservice such as resource consumption, REST-API calls or Failure counts. Metrics along with distributed tracing and distributed logging, provide the necessary data, operators need to analyze failures in services, which occurred in service call chains. The services must provide and publish metrics, which can be made available to a central metric service.

7.2.6 Fault Tolerance

The MicroProfile specifications provide the specification MicroProfile-Fault-Tolerance, which provides an API to define fault tolerance behavior such as retries, timeouts and error fall-backs. The fault tolerance of a service means, that if a depending service is not accessible at the time, a service must not fail immediately after the first try, but the service should retry to call the depending service for several times, and fail when all retries have failed. Such a behavior ensures that short timed communication errors, redeployments or overloads do not immediately cause a service to fail. The services must provide proper fault tolerance configuration and fall-back behavior to be able to recover from such errors in a proper manner. [Ecl18b].

7.2.7 API Management

The API management of a public API such as REST-API and REST-Models ensure that the clients, using a public API, are not broken by changes made on that API. There are several opinions on how API management can be done. A public API has to be stable per design, and needs to evolve and provide backward compatibility in a way, so that clients have enough time to catch up with the changes. Swagger has become very popular for documenting REST-API, where the documentation can be used to generate clients, provide documentation for developers and to test the public API. The services must be capable of migrating their public API in a way that the clients are not broken by made changes and need to publish the Swagger definitions of their public REST-API. [Par18; Sma18].

7.3 Openshift Architecture

Figure 7.3 illustrates the design of the Openshift Project, which will host the integration services. As discussed in Section 5.2 on page 21, Openshift isolates the namespaces, by bringing in the concept of an Openshift Project. Services hosted in an Openshift Project which are not exposed via an Openshift Route, are implicitly protected from external access from the Internet or services hosted in other Openshift Projects. The Openshift Project will contain the Application Service, the Database Service and the Integration Service. The Application Service has access to the Internet and will be accessed by the Client from the Internet via its public address. The Integration Service and the Database Service are not exposed to the Internet and can only be accessed within the Openshift Project by their service names.

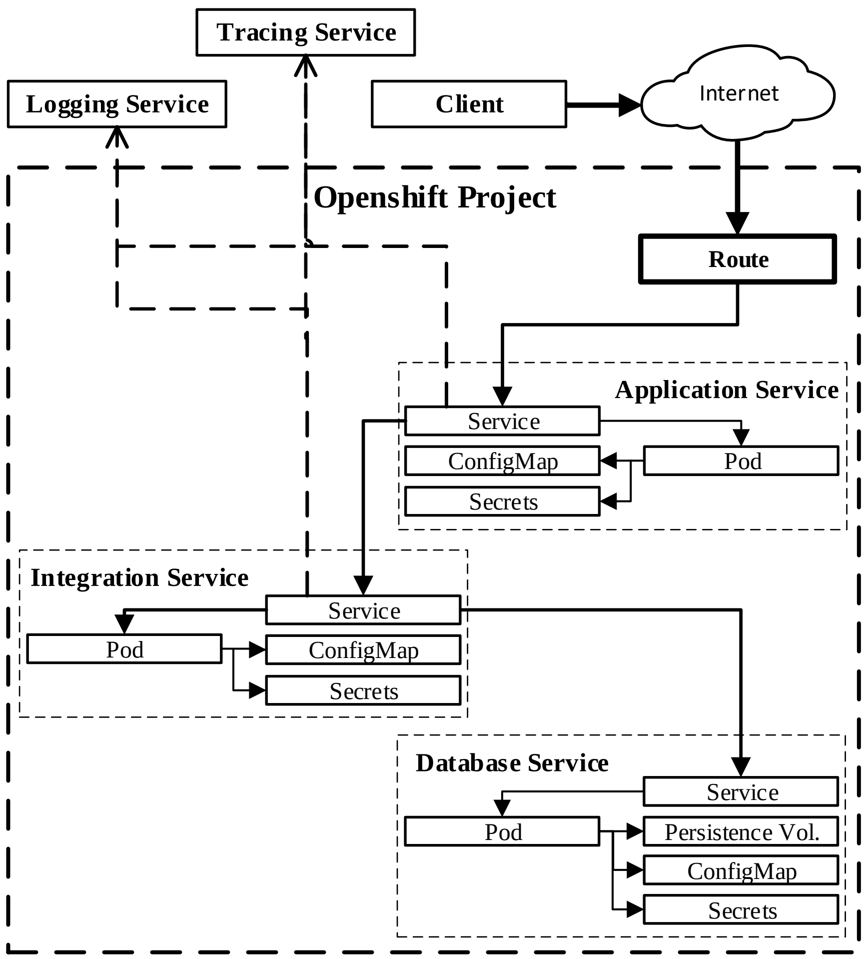

Figure 7.3: Openshift Project architecture

The orchestration and mediation of the services are managed by Openshift, whereby the services communicate with each other via their service names, which ensures that the communication stays within the Openshift Project. If multiple service instances are present, then Openshift handles the request assignment via Round Robin. Openshift will provide the configurations and secrets referenced by the integration service templates to the services as configured.

7.4 Openshift Requirements

The implementation of the Openshift resources such as templates and scripts must be implemented under consideration of the principles of IaC, which has been discussed in Section 2.2 on page 5. Keeping to the principles of IaC will ensure, that the Openshift Project can effortlessly be managed.

The Openshift Project will host the integration services and manage their configuration and secrets via Openshift ConfigMaps and Openshift Secrets, which have been discussed in Section 4.2.1 on page 16. The configurations and secrets are managed outside the microservice code bases and are provided by the hosting Openshift Project for a specific stage the Openshift Project represents. The configurations and secrets are supposed to be managed by operators and are only made available to the services during runtime and cannot be accessed by developers.

The integration services shall manage their integration into Openshift, by providing the necessary templates, whereby configurations and secrets are referenced by their predefined names and keys. The integration service must expose all necessary configuration parameters, which allow to configure the integration service for a specific stage a Openshift Project represents.

For demonstration purpose the Openshift Project hosting the integration services, shall host a tracing service and a log aggregation service as well. This is due to the fact, that in shared environments only HTTP(S) protocol is allowed, but tracing and logging data is usually send via UDP protocol.

The next Chapter 8 on the following page will discuss the implementation of the in this chapter designed prototype.

Chapter 8

Implementation ESB in Openshift

This chapter will discuss the implemented prototype, which has been designed in Chapter 7 on page 32. The implemented prototype uses a lot of technologies and specifications, which are beyond the scope of tis thesis, therefore this thesis will focus on the most important implementation parts. A focus will be set on the implementation of the aspects discussed in Section 7.2 on page 33, which ensure that a integration service can be managed properly in a Openshift Cluster.

The integration services are implemented as microservice, which run as standalone applications on the Openshift Cluster, with their own life cycle. The integration services communicate via REST with each other, whereby each service provides a proper managed public API. The code bases of the integration services are managed separately, which completely de-couples the integration service from each other.

As the prototype illustrates, the ESB is now represented by Openshift, which acts as the platform for the hosted integration services. The hosted integration services are implemented as microservices and are running in Docker Containers as standalone applications. Horizontal scaling and the distribution of the services over multiple hosts are now possible. Section 8.8 on page 47 will discuss the implemented Openshift resources, which are used to define and manage the Openshift Project.

Is is assumed, that the reader is familiar with Java Enterprise Development, Maven and microservices. The implemented prototype is available on Github [1]. The repository contains a README file, which describes how to setup the prototype.

8.1 Microservice Technologies

The following sections will give a brief introduction about the most important technologies used to implement the integration services. Each implemented service uses the same technologies and is build the same way, because no matter what the concrete purpose of the service is, they all have to be integrated and run the same way on a Openshift Cluster.

[1] https://github.com/cchet-thesis-msc/prototype

8.1.1 JBoss Fuse Integration Services 2.0

JBoss Fuse Integration Services 2.0 is a set of tooling for developing integration services running on a Openshift Cluster. It provides Openshift integrations for different frameworks such as Spring Boot, Karaf or Camel. The services are started via an Java-Agent such as Prometheus or Jolokia, which are used to monitor the service during runtime. Additionally to provided Openshift resources, a Maven Plugin is provided, which allows to interact with the Openshift Cluster during a Maven build. JBoss Fuse Integration Services 2.0 allows developers to interact with a Openshift Cluster in a way like developers did before with an application server [Huß18; Pro18].

8.1.2 Wildfly Swarm

Wildly Swarm is the JEE answer to Spring Boot, and is a framework, which allows to package an application into an Uber-JAR. An Uber-JAR is a packaged standalone application, which can be started with the command `java -jar`. During the packaging, only those components of an application server are packaged, which are referenced and needed by the application. The application can then be started via `java -jar app.jar`, whereby the application server is bootstrapped programmatically. The application deployment artifact is a JAVA Web-Archive, which could be hosted in any application server environment, which provides all of the referenced dependencies [Red18e].

8.1.3 Fabric8

Fabric8 is an integrated development platform for developing applications on Kubernetes. Fabric8 provides the Maven Plugin for the JBoss Fuse Integration Services 2.0, and focuses on building Docker Images, managing Kubernetes or Openshift resources and deploying Java applications on Kubernetes or Openshift Clusters [Red18b].

The next Section will discuss the implementations of the microservice aspects discussed in Section 7.2 on page 33.

8.2 Security

The integration services are secured with OAuth, and authenticate their clients via Keycloak. Keycloak is used as the authentication service, and is a very popular open source identity and authentication application. Wildfly Swarm provides an integration into Keycloak via the Keycloak-Adapter, which only needs to be added as a dependency to the Maven POM and to be configured.

8.2.1 Service Implementation

This section will discuss the implementation of the security in the service implementations. Listing 2 on the next page shows the dependency, which brings in the Keycloak Adapter, integrates itself into the Java Web-Security mechanisms, and can therefore be configured with Java Web-Security security constraints.

```xml
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>keycloak</artifactId>
</dependency>
```

Listing 2: Wildfly Swarm Keycloak dependency in pom.xml

Listing 3 shows an excerpt of the Wildfly Swarm configuration file project-stages.yml, which configures the security constraints for the REST-Endpoint.

```yaml
swarm:
  deployment:
    ${project.artifactId}.war:
      web:
        login-config:
          auth-method: "KEYCLOAK"
        security-constraints:
          - url-pattern: "/rest-api/*"
            roles: "[client]"
```

Listing 3: Security configuration in project-stages.yml

The following two listings are excerpts of the deployment.yml Openshift Template, which is managed in the service code base. Listing 4 shows the specification of the secret injection into a Docker Volume. The secrets are injected as files, whereby the file name represents the secret key and the file content represents the secret value. Therefore, that the secrets are managed externally, the developers need to provide the secret name for the service deployment configuration. In this case an expression is used, which can be replaced by Maven Properties, whereby the Maven Properties can be provided in the pom.xml or provided/overwritten by Java Options, during the build process.

```yaml
template:
  spec:
    volumes:
      - name: "app-config"
        secret:
          secretName: "${oc.secret-service-app}"
```

Listing 4: Configuration of the secret injection

Listing 5 on the following page show the specification of the mount of the Docker Volume, which provides the secrets. The mount path is also represented by a Maven Property, because this path is also used in the project-stages.yml file, where it points to the service configuration source for the productive stage. The secrets consumed by the services are used the same way as non-sensitive configurations, which are discussed in Section 8.3 on the next page.

```
containers:
  - name: "${project.artifactId}"
    volumeMounts:
      - name: "app-config"
        mountPath: "${oc.secret-service-app.dir}"
```

Listing 5: Configuration volume mount

8.2.2 Openshift Implementation

This section will discuss the Openshift implementation, whereby the implementation is represented by a shell script, which manages the secrets. The secrets are managed outside the code bases of the integration services.

Listing 6 shows the Openshift CLI-Commands, which are used to create the secrets. The first command creates a secret from literal values which is used by a client service to retrieve authentication tokens. The second command creates a secret from a file, whereby the filename is the secret key and the content is the secret value, which is used by the Keycloak Adapter to validate authentication tokens provided by the clients.

```
oc create secret generic secret-service-app \
   --from-literal="service.db.base-url=${SERVICE_BASE_URL}"
   --from-literal="keycloak.token-url=${SERVICE_AUTH_URL}"
   --from-literal="keycloak.client.id=${SERVICE_CLIENT_ID}"
   --from-literal="keycloak.client.secret=${SERVICE_CLIENT_SECRET}"

oc create secret generic "${SECRET_SERVIVE_KEYCLOAK}"
   --from-file="./keycloak.json"
```

Listing 6: Openshift CLI command for creating the secret

The discussed implementations are the necessary implementations on the Service and Openshift side, to secure services hosted on a Openshift Cluster. No infrastructure code is necessary, only configuration. The following section will discuss the configuration of the integration services, which can be applied to the security as well, because secrets in Openshift are used in the service implementations the same way as configuration parameters.

8.3 Configuration

The integration services use the MicroProfile Config-API to be configurable for multiple stages by exposing all necessary configuration parameters and to be able to consume configuration parameters from different configuration sources via injection. Developers are bound to the configuration or secret name, keys and their value type, but developers are not bound to the configuration or secret source, which allows to provide configurations from different sources and for different stages.

8.3.1 Service Implementation

This section will discuss the implementation of the configuration definition and usage. Listing 7 shows the dependency, which brings in the MicroProfile Config-API and implementations to enable injectable configurations.

```xml
<dependency>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>microprofile-config</artifactId>
</dependency>
```

Listing 7: Wildfly Swarm MicroProfile-Config dependency in pom.xml

Listing 8 shows the definition of the configuration source with the name *app.secrets* for the development stage, whereby the configuration properties are provided hard coded.

```yaml
project:
  stage: "dev"
swarm:
  microprofile:
    config:
      config-sources:
        app.secrets:
          properties:
            service.db.base-url: "http://localhost:8080/rest-api"
            keycloak.token-url: "http://localhost:9080/auth/token"
            keycloak.client.id: "client"
            keycloak.client.secret: "client-secret"
```

Listing 8: Hard coded configuration for development stage

Listing 9 shows the definition of the configuration source with the name *app.secrets* for the production stage, whereby the configurations are loaded via a directory. The directory location is represented by an Maven Property, because its used in multiple configuration files, as already discussed in Section 8.2.1 on page 39. The MicroProfile Config-API and implementations will load files in this directory by using the filename as the key and the content as the value.

```yaml
project:
  stage: "prod"
swarm:
  microprofile:
    config:
      config-sources:
        app.secrets:
          dir: "${oc.secret-service-app.dir}"
```

Listing 9: External configuration for production stage

Listing 10 shows the injection of the Keycloak Secrets into a CDI Bean, whereby the actual configuration source, the configurations are retrieved from, is unknown. The injected configuration properties are retrieved from a Openshift Secret, but are used in source code the same way as configurations. The Keycloak Secrets are used to retrieve authentication tokens for the REST calls.

```
@Inject
@ConfigProperty(name = "keycloak.token-url")
private String keycloakTokenUrl;

@Inject
@ConfigProperty(name = "keycloak.client.id")
private String keycloakClientId;

@Inject
@ConfigProperty(name = "keycloak.client.secret")
private String keycloakClientSecret;
```

Listing 10: Injection of Keycloak configuration parameters

8.3.2 Openshift Implementation

The Openshift implementation has already been covered by Section 8.2.2 on page 41, because configurations and secrets are injected into Docker Containers the same way, and the configuration shown in this section are actual retrieved from a Openshift Secret.

8.4 Tracing

The integration services use the MicroProfile OpenTracing-API to provide tracing data to a central tracing service. Jaeger[2] is used as the tracing service, which collects all tracing data and provides a GUI for analyzing traces.

8.4.1 Service Implementation

This section will discuss the implementation of the service tracing. Listing 11 shows the dependency, which brings in the MicroProfile OpenTracing-API and implementations to enable tracing.

```
<dependency>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>opentracing</artifactId>
</dependency>
```

Listing 11: Wildfly Swarm MicroProfile-OpenTracing dependency in pom.xml

[2] https://www.jaegertracing.io/

Listing 12 shows the configuration for the integration into an external tracing service, whereby the configuration parameters are provided by Maven Properties, environment variables and constants. The configuration properties are created as System Properties by Wildfly Swarm, whereby expressions like *${env.JAEGER_PORT}* are resolved during startup.

```
JAEGER_SERVICE_NAME: "${project.build.finalName}"
JAEGER_AGENT_HOST: "${env.JAEGER_HOST}"
JAEGER_AGENT_PORT: "${env.JAEGER_PORT}"
JAEGER_REPORTER_LOG_SPANS: "true"
JAEGER_REPORTER_FLUSH_INTERVAL: "1000"
JAEGER_SAMPLER_TYPE: "const"
JAEGER_SAMPLER_PARAM: "1"
```

Listing 12: Configuration for integration into tracing service

Listing 13 shows a class which is annotated with `@Traced` on class level, which enables tracing for all methods within this class. The annotation `@Traced` enables an interceptor, which implements the tracing logic.

```
@Traced
public class ReportServiceImpl implements ReportService {
   ...
}
```

Listing 13: Enable tracing for a class

A trace is a set of so called spans, whereby a span represents one call in a call chain and contains meta-data of the call such as call duration. The interceptor creates a new span for each call and appends the created span to an existing parent span, or the created span is the parent span.

8.4.2 Openshift Implementation

The communication between the integration services and tracing service is done via UDP protocol, and therefore Openshift does need any special configuration.

8.5 Logging

The integration services provide logging to a central log aggregation service. Graylog[3] is used as the log aggregation service, which collects all logging data and provides a GUI for analyzing logs.

[3] https://www.graylog.org/

8.5.1 Service Implementation

This section will discuss the implementation of the service logging. Listing 14 shows the dependencies, which bring in the logging implementations. SLF4J[4] has been chosen as the logging facade, whereby an integration in Wildfly Swarm used JBoss Logging is provided by SLF4J.

```
<dependency>
    <groupId>org.wildfly.swarm</groupId>
    <artifactId>logging</artifactId>
</dependency>

<dependency>
    <groupId>org.jboss.slf4j</groupId>
    <artifactId>slf4j-jboss-logging</artifactId>
    <version>${slf4j.jbopss-logging.version}</version>
</dependency>
```

Listing 14: Wildfly Swarm logging dependencies in pom.xml

The following listings are part of the project-stages.yml configuration file and configure logging for different stages. Listing 15 shows the configuration of the logging format, which uses Mapped Diagnostic Context (MDC) parameters to mark a log entry with the transaction id. The configured formatter is used over all stages.

```
swarm:
  logging:
    pattern-formatters:
      DEFAULT_LOG_PATTERN:
        pattern: "%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p (%t) [%C{2}] \
                  transaction.id=%X{transaction.id} - %m%n"
    console-handlers:
      CONSOLE:
        named-formatter: "DEFAULT_LOG_PATTERN"
```

Listing 15: Configuration of the logging format

Listing 16 shows the logging configuration for the development stage.

```
swarm:
  logging:
    root-logger:
      level: "DEBUG"
      handlers:
        - "CONSOLE"
```

Listing 16: Configuration of the logging for development stage

[4]https://www.slf4j.org/

Listing 17 shows the configuration of the logging for the production stage, where the service is contributing its logs to a central log aggregation service. A Syslog logging handler is configured, which sends the logs to the log aggregation service via the SYSLOG[5] protocol.

```
swarm:
  logging:
    custom-handlers:
      SYSLOGGER:
        named-formatter: "DEFAULT_LOG_PATTERN"
        attribute-class: "org.jboss.logmanager.handlers.SyslogHandler"
        module: "org.jboss.logmanager"
        properties:
          serverHostname: "graylog"
          hostname: "${project.build.finalName}"
          port: "10514"
          protocol: "UDP"
    root-logger:
      level: "WARN"
      handlers:
        - "CONSOLE"
        - "SYSLOGGER"
```

Listing 17: Configuration of the logging for production stage

Listing 18 shows the implementation of the interface *ContainerRequestFilter*, which is used to caputre the trace transaction id on a REST Endpoint. The implementation is depending on the Uber MicroProfile OpenTracing-API implementation, because the specification does not provide an transaction id yet.

```java
@Provider
public static class MDCContainerRequestFilter
  implements ContainerRequestFilter {

  @Inject
  private Instance<Scope> scopeInstance;

  @Override
  public void filter(ContainerRequestContext requestContext)
    throws IOException {
          // Uber specific format 'aaa:ffff:0:1'
          final String tracingId = scopeInstance.get().span()
                                   .context().toString()
                                   .split(":")[0];
          MDC.put("transaction.id", tracingId);
  }
}
```

Listing 18: Capture of tracing id on REST Endpoint

[5]https://tools.ietf.org/html/rfc5424

Listing 19 shows the CDI Producer for the logger instance. The logger is produced for the Dependent Scope, which means that the producer method gets called for each injection point of each CDI Bean.

```
@ApplicationScoped
public class LoggerConfiguration {

    @Produces
    @Default
    @Dependent
    Logger createLogger(final InjectionPoint ip) {
        if (ip.getBean() != null) {
            return LoggerFactory.getLogger(ip.getBean()
                                  .getBeanClass());
        } else if (ip.getMember() != null) {
            return LoggerFactory.getLogger(ip.getMember()
                                  .getDeclaringClass());
        } else {
            return LoggerFactory.getLogger("default");
        }
    }
}
```

Listing 19: Logging instance CDI Producer

8.5.2 Openshift Implementation

8.6 Fault Tolerance

8.6.1 Service Implementation

8.6.2 Openshift Implementation

8.7 API Management

8.7.1 Service Implementation

8.7.2 Openshift Implementation

8.8 Openshift Project

8.8.1 Configuration

8.8.2 Secrets

Chapter 9

Evaluation ESB in Openshift

9.1 Software Development

9.2 Release Management

9.3 Security Handling

9.4 API Versioning

9.5 System Resources

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).

[HLA05] Colombe Herault, Philippe Lalanda, and Equipe Adele. "Mediation and Enterprise Service Bus A position paper" (2005) (cit. on p. 27).

[HW08] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. 11th ed. Boston, MA: Pearson Education, Inc., 2008 (cit. on pp. 1, 26).

[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+15a] Lawrence Liu et al. *Integration Platform as a Service: The next generation of ESB, Part 1*. IBM, 2015 (cit. on pp. 1, 20, 28).

[Liu+15b] Lawrence Liu et al. *Integration Platform as a Service: The next generation of ESB, Part 2*. IBM, 2015 (cit. on pp. 20, 28).

[Lop08] Lopez-Sanz, Marcos and Acuna, Cesar J. and Cuesta, Carlos E. and Marcos, Esperanza. "Modelling of Service-Oriented Architectures with UML" (2008) (cit. on p. 31).

[Mas18] Masternak, Tomasz and Psiuk, Marek and Radziszowski, Dominik and Szydd, Tomasz and Szymacha, Robert and Zieliski, Krzysztof and Zmuda, Daniel. "ESB-Modern SOA Infrastructure" (June 2018) (cit. on p. 27).

[Red05] Red Hat Inc. "Digital Invocation Through Agile Integration" (2005) (cit. on p. 26).

[Ric15] Richards, Mark. *Microservices vs. Service-Oriented Architecture*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2015 (cit. on p. 31).

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

Online sources

[Ama18a] Amazon Web Services (AWS). *Amazon EKS*. 2018. URL: https://aws.amazon.com/eks/ (visited on 03/31/2018) (cit. on p. 15).

[Ama18b] Amazon Web Services (AWS). *AWS Elastic Beans Talk*. 2018. URL: https://aws.amazon.com/elasticbeanstalk/ (visited on 04/12/2018) (cit. on p. 20).

[Clo18a] 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).

[Clo18b] Cloud Native Computing Foundation. *Vendor-neutral APIs and instrumentation for distributed tracing*. 2018. URL: http://opentracing.io/ (visited on 06/05/2018) (cit. on p. 35).

[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.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. *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).

[Ecl18a] Eclipse Foundation. *Configuration for MicroProfile*. 2018. URL: https://github.com/eclipse/microprofile-config (visited on 06/05/2018) (cit. on p. 34).

[Ecl18b] Eclipse Foundation. *Fault Tolerance*. 2018. URL: https://github.com/eclipse/microprofile-fault-tolerance (visited on 06/05/2018) (cit. on p. 35).

[Goo18a] Google Cloud. *Google App Engine*. 2018. URL: https://cloud.google.com/appengine/ (visited on 04/12/2018) (cit. on p. 20).

[Goo18b] 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/Documentation/cgroup-v2.txt (visited on 03/30/2018) (cit. on p. 13).

[Huß18] Huß, Roland. *Jolokia JMX on Capsaicin*. 2018. URL: https://jolokia.org/ (visited on 06/11/2018) (cit. on p. 39).

[Kub18a] Kubernetes. *API Overview*. 2018. URL: https://kubernetes.io/docs/reference/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/workloads/pods/pod/ (visited on 04/01/2018) (cit. on p. 17).

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

[Kub18e] Kubernetes. *Services*. 2018. URL: https://kubernetes.io/docs/concepts/services-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://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (visited on 03/30/2018) (cit. on p. 13).

[Mic18a] Microsoft Azure. *Azure Cloud Services*. 2018. URL: https://azure.microsoft.com/en-us/services/cloud-services/ (visited on 04/12/2018) (cit. on p. 20).

[Mic18b] Microsoft Azure. *Azure Container Service (AKS)*. 2018. URL: https://docs.microsoft.com/en-us/azure/aks/ (visited on 03/31/2018) (cit. on p. 15).

[oau18] oauth.net. *An open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications*. 2018. URL: https://oauth.net/ (visited on 06/08/2018) (cit. on p. 34).

[Ope14] Openshift. *OpenShift V3 Deep Dive Tutorial | The Next Generation of PaaS - Archived*. 2014. URL: https://blog.openshift.com/openshift-v3-deep-dive-docker-kubernetes/ (visited on 04/05/2018) (cit. on p. 21).

[Ope18a] Openshift. *App Development with OpenShift Online*. 2018. URL: https://www.openshift.com/get-started/index.html (visited on 04/12/2018) (cit. on p. 20).

[Ope18b] Openshift. *OpenShift Application Platform*. 2018. URL: https://github.com/openshift/origin (visited on 04/12/2018) (cit. on p. 21).

[Ope18c] Openshift. *Overview*. 2018. URL: https://docs.openshift.com/container-platform/3.5/architecture/core_concepts/ (visited on 04/07/2018) (cit. on p. 21).

[Ope18d] Openshift. *Source-To-Image (S2I)*. 2018. URL: https://github.com/openshift/source-to-image (visited on 04/07/2018) (cit. on p. 24).

[Par18] Paraschiv, Eugen. *Versioning a REST API*. 2018. URL: http://www.baeldung.com/rest-versioning (visited on 07/02/2018) (cit. on p. 35).

[Pro18] Prometheus. *From metrics to insight*. 2018. URL: https://prometheus.io/ (visited on 06/11/2018) (cit. on p. 39).

[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. *Get Started with Red Hat JBoss Fuse 7 Tech Preview 3 Today!* 2018. URL: https://developers.redhat.com/blog/2018/02/27/red-hat-jboss-fuse-7-tech-preview/ (visited on 04/12/2018) (cit. on p. 20).

[Red18b] Red Hat Inc. *Integrated Development Platform*. 2018. URL: http://fabric8.io/ (visited on 07/02/2018) (cit. on p. 39).

[Red18c] 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).

[Red18d] 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).

[Red18e] Red Hat Inc. *Rightsize your Services*. 2018. URL: http://wildfly-swarm.io/ (visited on 07/02/2018) (cit. on p. 39).

[Sma18] SmartBear Software. *Swagger*. 2018. URL: https://swagger.io/ (visited on 06/05/2018) (cit. on p. 35).

[The15] The Apache Software Foundation. *Apache Camel*. 2015. URL: http://camel.apache.org/ (visited on 06/30/2018) (cit. on p. 26).