Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

Implementation of an Enterprise Service Bus with OpenShift

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Ing. Thomas Herzog B.Sc.

Betreuer:	Dr. Erhard Siegl, Gepardec IT-Services GmbH, Wien
Begutachter:	DI (FH) Peter Kulczycki

September 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, September 7, 2018

Contents

Abstract

An Enterprise Service Bus (ESB) is a crucial part of an enterprise, which connects the enterprise to its partners, customers, and other branches. The appearance of containerization, cloud services and the microservice architecture have provided new possibilities for implementing and running an ESB. But, an ESB is commonly used by large conservative enterprises, which don't adapt new technologies fast, and wait until a new technology has proven itself. Especially the cloud is something the industry denied to use for a long time, because of the fact, that the infrastructure and data are managed and maintained by external service providers.

These days, we live in the so called cloud age, whereby global enterprises like Red Hat or Amazon provide cloud services such as Platform as a Service (PaaS), which can scale with the business. Enterprises start to consider to move their ESB installations to the cloud to profit from the cloud service provided features. Moving an ESB to the cloud will be a long term process for an enterprise, because the established processes for development, running, and managing the ESB, will have to change.

This thesis has the goal to give the reader an overview of the cloud related concepts and technologies such as, Infrastructure as Code (IaC), and Docker, which are the base for cloud services. The implemented ESB prototype, is available at https://github.com/cchet-thesis-msc/prototype, and shows how an ESB could be implemented on a PaaS platform.

Chapter 1

Introduction

1.1 Motivation

Large enterprises work with several independent applications, whereby each application covers an aspect of their business. 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 exchange data, which is commonly represented in different formats and versions. This leads to a highly heterogeneous network of applications, which is 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. Red Hat JBoss Fuse is be based on the JBoss Enterprise-Application-Platform (JBoss EAP), where all integration services run in the same runtime environment[Red18c].

With the appearance of cloud solutions, such as Platform as a Service (PaaS), it is now possible to move an ESB from a dedicated environment to a cloud environment, whereby each integration service runs in its own runtime environment, rather than joining an existing runtime environment. The concept of Integration Platform as a Service (IPaaS) is built on top of PaaS, and enhances a common PaaS solution with the integration features required 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. The cost are reduced due to the fact, that the ESB can scale down when its load decreases, whereby less resources are consumed, which have to be paid for.

1.2 Objectives

This thesis aims to implement an ESB on Openshift PaaS. An ESB is different to a Service Hub, because the ESB is a distributed system by design, whereby for instance, transformation is not centralized, as it is with an Service Hub. A Service Hub has a central component, which performs transformation, routing and orchestration, whereby the scaling is limited by the hardware capabilities of the central hub. Additionally, the hub component of a Service Hub represents a single point of failure[HW08].

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 the microservice architecture[Red18d].

Before implementing an ESB, which runs on a PaaS platform such as Openshift, it is necessary to understand the new concepts such as Infrastructure as Code (IaC) or containerization with Docker, which are covered in the following chapters. The microservice architecture and cloud services such as PaaS are becoming more important for the software industry, because of the features they provide. For instance, Red Hat is currently moving its ESB middleware JBoss Fuse to the cloud, whereby JBoss Fuse will fully rely on Openshift, and the integration services will have to be implemented as microservices. This has a 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, which 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 of their applications and little refactoring work on their application sources. It will be hard for them to reduce the accumulated 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.

The term “monolithic ESB” refers to an ESB implementation, whereby all integration services are part of a single application, and therefore, are part of the application life-cycle, instead of having an own life-cycle per integration service. An integration service is a service, which integrates two or more other service with each other, by handling aspects like data transformation, routing, or security, between the integrated services.

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. IaC descriptors are called templates, cookbooks, recipes, or playbooks, depending on the IaC tool. In the further course, the IaC descriptors 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[Kie16].

The term "system" is used as a general description, whereby in the context of IaC, a system can be anything, which can be described via a DSL. The term "environment" is also used as a general description for a habitat for one or more systems.

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. The IT systems are decoupled due to the fact, that cloud providers cannot allow their customer to tamper with the underlying hardware and operating system. In general, the hardware resources provided by a cloud provider are shared by multiple customers[Kie16].

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

The term “infrastructure” is used as a general description for the foundation of systems, which provides any kind of resources the systems can consume.

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, which are 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, which provide the infrastructure and the tooling. A well known problem, which enterprises will face, is the so called Automation Fear-Spiral, which is shown in Figure 2.1.

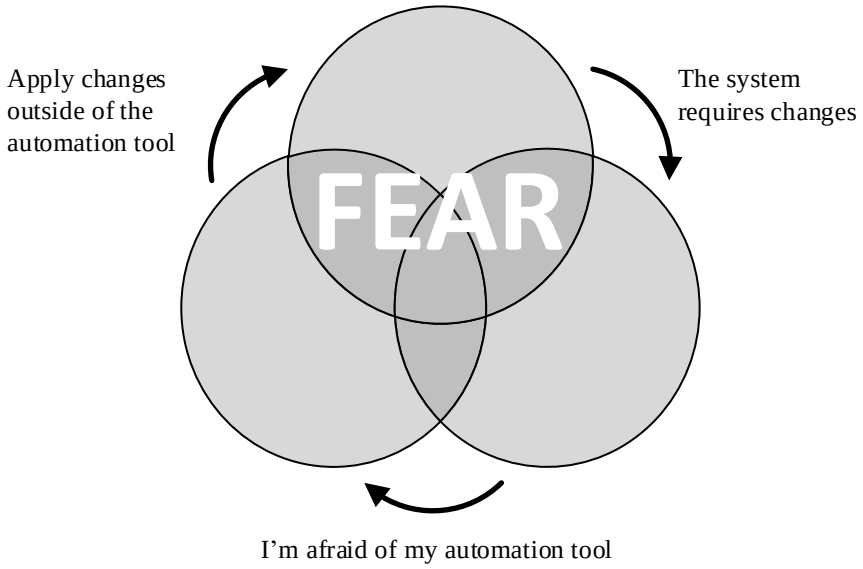

Figure 2.1: Automation Fear-Spiral

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 maybe 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, whereby the systems can become more complex, but still easy to manage, and 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 systems, the creation and maintenance of systems has become simple, due to the abstraction layer provided by the IaC DSL and tooling.

2.2.1 Systems are Reproducible

With IaC, systems are reproducible. It is possible to reproduce a whole system 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 systems effortlessly is a powerful capability, because it can be done automatically, consistently and with less risk of failures. The reproducibility of a system is based on reusable templates, which expose parameters, whereby the parameter values are defined for a specific environment. Figure 2.2 illustrates the reproduction of a system into two different environments, whereby the IaC tooling reads the templates and sets the defined parameters with the environment specific values[Kie16].

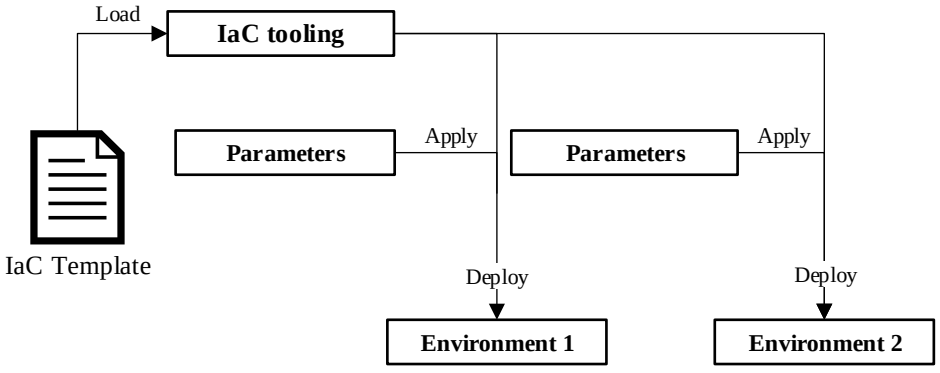

Figure 2.2: Schema of a parametrized system reproduction

2.2.2 Systems 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, describing a system, don't have to be applied on an existing system, but can be applied by destroying and recreating the system. A 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 the fact, that the system could change at any time. Systems must not fail, because a disposable system disappears and reappears again because of a redeployment [Kie16].

2.2.3 Systems are Consistent

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

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

```
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 of Docker Compose

2.2.4 Actions are Repeatable

Building reproducible systems means, that any action applied to the system has to be be repeatable. Without repeatability, the automation cannot be trusted and systems wouldn't be reproducible. An instance of a system in another environment has to 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 easy to manage systems in the cloud age. Enterprises can make use of IaC to make their existing systems more reliable, consistent and easier to manage. Nevertheless, before an enterprise can profit from IaC, it has to apply clear structures to their development process, as well as sticking to the automation process provided by the IaC tooling. For experienced administrators, who are used to maintain systems manually, it could be sometimes hard to understand why they are not supposed to perform any actions on the system manually anymore. 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, because they are abstracted from the technologies by the DSL and IaC tool.

The following chapter will discuss containerization with Docker, which is very popular tooling for isolating processes on a Linux host. Docker strongly relies on the principle of IaC, and therefore can be seen as a IaC tool, for managing isolated processes.

Chapter 3

Containerization with Docker

Docker is a tool for creating, provisioning and interacting with Linux Containers (LXC). 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 discussed in Section 3.3 on page 13. 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 on page 4. When using Docker, Linux Containers are often referred to as Docker Containers[Doc18e; Lin18a].

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 The need for Containerization

Containerization is a key factor for cloud platforms such as PaaS, where each application runs in its own isolated container. A container is an instance of an image, which represents the initial state of an application. A VM represents a full blown OS, whereby the OS provides a kernel, which is emulated by the Hypervisor on the host OS. 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 a kernel 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, whereby the containers run on the provisioned VMs. The usage of containers compared to the usage of VMs can reduce costs for hosting applications. Applications running in containers have less resource impact compared to applications running 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 host[Doc18b].

When enterprises have applied IaC to their systems, then the next logical step is to apply IaC to their applications as well. Applications running in containers profit from the IaC principles immutability, reproducibility, repeatability, and consistency, which have been discussed in Section 2.2 on page 6. Docker can be seen as an IaC tool, whereby the Dockerfiles represent the IaC templates and the Docker CLI acts as the IaC tool.

When using Docker, developers define the process environment for their applications via a Dockerfile, and system administrators provide a VM with the Docker Engine, which is a shift of responsibility. Thus, system administrators have no control about the application environment anymore, which is not considered to be a problem, because the application processes are isolated from other processes, and cannot interfere them. 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 will give an overview of the Docker technology, its architecture and artifacts.

3.2 Docker

Docker is an open source software, which instruments kernel features like LXC. It provides a community edition for free, as well as an enterprise support for production use. 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 publish Docker Images[Doc18c].

3.2.1 Docker Engine

Figure 3.1 on the following page 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. The Docker Engine was initially designed for LXC exclusively, but has been ported to Windows as well. Docker Images and Docker Containers, which were created for a 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.

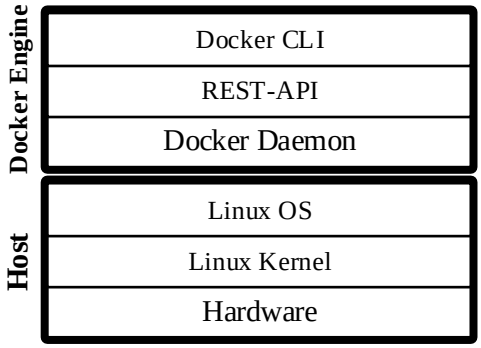

Figure 3.1: Docker Engine architecture

Docker Daemon

The Docker Daemon represents the background process, which creates, runs and manages the Docker Containers on the Docker Host, and is similar to a VM Hypervisor, but only on process level. 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 the Docker Engine exposed REST API, because the Docker Engine is designed as a client server 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. All commands executed via the Docker CLI are actually sent to the Docker Engine via its exposed REST API.

Docker Command Line interface

The Docker Engine provides a Docker Command-Line-Interface (Docker CLI) for interacting with the Docker Daemon via a Linux shell. The Docker CLI itself is nothing more than a wrapper for the REST API, and 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 container resources such as volumes and networks, managing 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 immutable read-only layers, whereby one layer is created for each instruction in the Dockerfile. A layer can contain meta data or a file-system state. Docker Images are hierarchical and can inherit from another Docker Image, which is then called a 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 Images can be identified via their tags, whereby a Docker Image-Tag has the form of *[url]<namespace>/<name>:<version>* e.g. *docker.io/library/openjdk:8-alpine*.

Docker Containers

A Docker Container is an instance of a Docker Image, whereby a new writable layer is appended, which contains all new or modified files. Deleting a Docker Container means, deleting the appended writable layer. Two Docker Containers, created from the same Docker Image, have their own appended writable layer, and don't share any resources. A Docker Container keeps running as long as the contained foreground process is running. Without a foreground process, the Docker Container stops immediately, after the command was executed. The process running in the Docker Container is completely isolated from other processes, and is only allowed to use its assigned resources.

Docker Volumes

There are two ways to keep data persistent in Docker Containers. The first one is, to mount a host file-system into the Docker Container, whereby the Docker Container depends on the specific host file-system. The second one is, to use a Docker Volume, which abstracts the underlying storage type from the Docker Container, and is backed by a storage driver. Cloud storage providers can implement a driver, so that the data, stored in a Docker Volume, is actually kept persistent on a cloud storage. Another advantage of Docker Volume is, that the Docker Volumes can be moved between OS, which support Docker, and allow more detailed configuration.

Docker Network

Docker provides networking features, whereby Docker Containers can be connected to each other via a Docker Network. Docker Networks are an abstraction of the actual networks, and are backed by a driver, which implements the actual network type. The Docker Containers are not aware of the actual network type, the same as they are not aware of the actual storage type with Docker Volumes.

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, whereby the Docker Client can be located on a remote host or on the Docker Host. The Docker Host runs the Docker Engine, which exposes the REST API, the Docker Client connects to. The Docker Engine manages the Docker Images and Docker Containers located on the Docker Host. The Docker Engine can push/pull Docker Images to/from a local/remote Docker Registry.

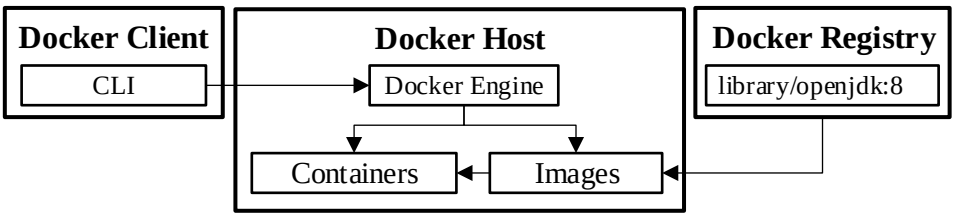

Figure 3.2: Docker Architecture

3.2.3 Docker Machine

The Docker Machine is a tool for managing local or remote Docker Hosts. With Docker Machine, an administrator can manage multiple Docker Hosts via a management server, without the need to connect to the Docker Host via a secure shell (SSH). The Docker Machine CLI provides all necessary commands for managing Docker Hosts. The Docker Engine provisions Docker Containers on a Docker Host and the Docker Machine provisions Docker Hosts on remote Linux 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[Doc18a].

3.3 Virtualization vs. Containerization

Before LXC, the industry made use of OS virtualization, to isolate their applications. A VM is managed by a Hypervisor, which is a 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, a VM represents a full blown OS, which itself has a resource impact, which adds to the resource impact of the running application. LXC on the other hand is a kernel technology, which provides resources such as network and storage to the application as well, but these resources are managed by the host OS, where the applications are running on.

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 applications running on a virtualization system.

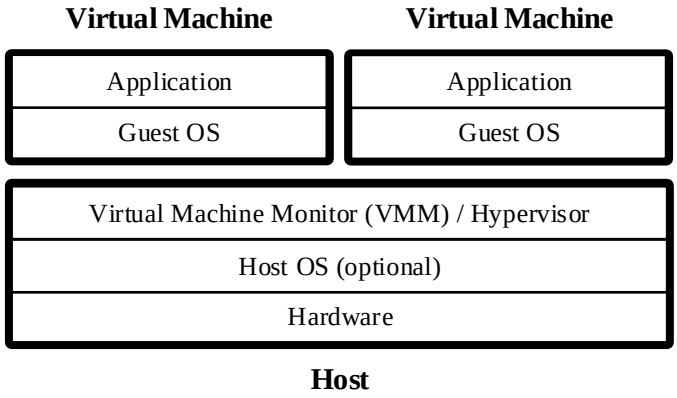

Figure 3.3: Architecture of virtualized applications

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. 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[Cos12].

3.3.2 Linux Container

The appearance of LXC has eliminated the shortcoming of the Linux OS, which wasn't able to isolate applications properly, which lead to OS virtualization for application isolation. 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 on the next page, 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 all available system resources, whereby other processes would starve.

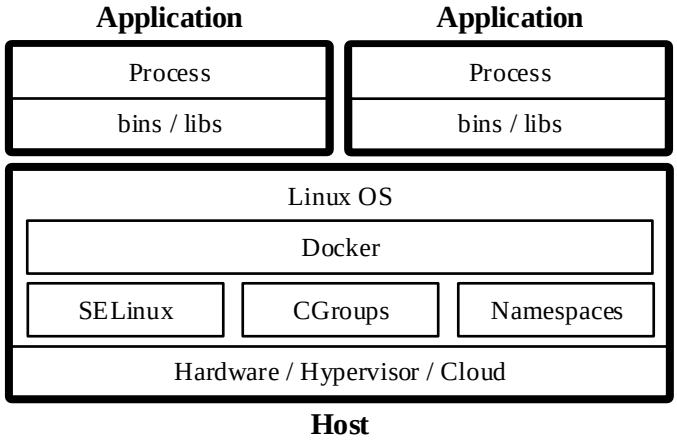

Figure 3.4: Architecture of containerized applications

The following sections discuss the two most important kernel features for LXC, Cgroups, and Namespaces, which provide resource control and isolation.

Cgroups

Cgroups stands for control groups and Cgroups provide the ability to aggregate processes, their child processes, and threads within theses processes, to groups, which are managed in a tree structure. Each group gets a portion of the global resources such as CPU cycles or memory assigned, whereby it is guaranteed, that a group and its managed processes cannot consume more resources as the group has been assigned to. Each process running in a container is assigned to a group, whereby a process cannot steal resources from another process anymore, because the resource assignments of a 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 its own namespace, and therefore the container has a limited view of the system, such as networks and process ids (PIDs). Namespaces are a fundamental concept of LXC, and Namespaces provide the isolation between containers[Cor14; Lin18b].

The following chapter discusses Kubernetes, which is an orchestrator for Docker Containers, and allows to manage Docker Containers in a clustered environment.

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, whereby 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. Kubernetes has been invented by Google, is open source since 2015, and managed by the Cloud Native Computing Foundation (CNCF), whereby CNCF 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 face the need to dynamically adapt to workloads and to roll out new version of their services fast, and without any downtime. Applying dynamically to workloads requires a dynamic infrastructure, which is capable of scaling up, when the workload increases, and scaling down, when the workload decreases, which is non trivial to be handled manually. Rolling out new versions without downtime require a well defined workflow, which ensures a well defined roll out behavior. For such uses cases, a CaaS platform like Kubernetes can be used. 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 provides a DSL, which allows to specify the desired state of the Ku-

bernetes Cluster, as well as a automation tooling to interact with the Kubernetes Cluster. 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 a platform for their services, which is effortlessly to specify and maintain via templates, and the provided automation tooling, which makes Kubernetes an IaC tool as well, as discussed in Chapter 2 on page 4. This makes it easy to modify the infrastructure at any time, which allows enterprises to apply to new requirements fast.

4.2 Kubernetes

Kubernetes is a CaaS platform to orchestrate containers in a cluster, whereby the Kubernetes Cluster-Nodes can be located in the cloud or in a dedicated data center. Kubernetes is designed as a client server architecture and a master slave architecture. At least one node in the Kubernetes Cluster acts as the Kubernetes Master, which is discussed in Section 4.2.2 on the next page, and the other nodes in the Kubernetes Cluster act as the Kubernetes Workers, which are discussed in Section 4.2.3 on page 20. The Figure 4.1 illustrates the architecture of a Kubernetes Cluster.

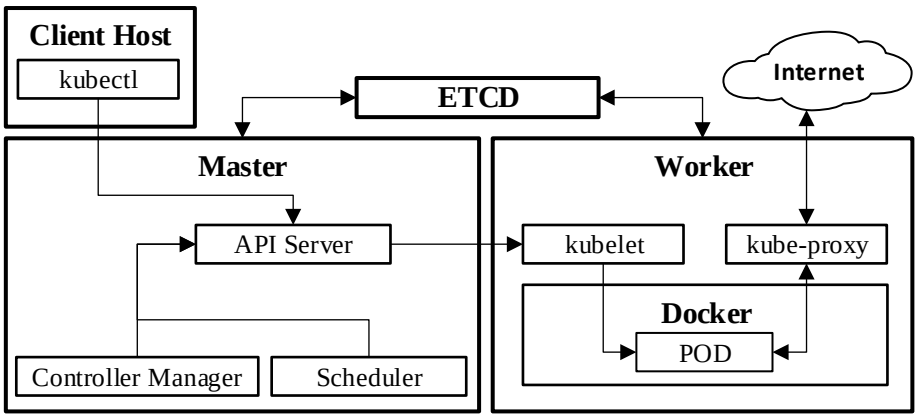

Figure 4.1: Architecture of a Kubernetes Cluster

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 state of the Kubernetes Objects, and the Kubernetes Clusters itself will ensure that the modified state is applied on the Kubernetes Cluster. The following sections will briefly introduce 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, and therefore share the same life-cycle. 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-Node, 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, whereby the containers in a single Pod represent one instance of one application[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-Node 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 load balance the request between the Pods of the replica set, depending on the chosen algorithm[Kub18g].

Secret

A secret is an abstraction to manage sensitive configuration data, which can be consumed by containers. A secret holds a set of key value pairs, which represents the sensitive configuration data, and can be referenced by its name. A referenced secret can be injected into the container, either as a environment variable, or a file, whereby the filename will represents the secret key and the file content represents the secret value. Developers reference secrets in the specifications by their name, and only the referencing containers can access the injected sensitive configuration data the secret holds[Kub18e].

ConfigMap

A configuration map works the same way as secrets, but is intended to hold non sensitive configuration data[Kub18f].

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 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 the 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 nodes and the Kubernetes Worker nodes. The Kubernetes Master provides configuration for the Kubernetes Worker nodes, and the Kubernetes Worker nodes provide state information for the Kubernetes Master nodes[Cor18].

Kubernetes API-Server (kube-apiserver)

The Kubernetes API-Server exposes the REST API for interacting with the Kubernetes Cluster, and is located on the Kubernetes Master. It represents the front-end of the Kubernetes Cluster and provides all necessary API to manage the Kubernetes Cluster, and the Kubernetes Objects.

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 managing 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 runs 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 used container runtime. 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-Node, which interacts with the Kubernetes Master via its exposed REST API. 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 back-ends. Any communication of a Pod to another Pod or to an external network 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 mostly Docker is used, which has been discussed in Chapter 3 on page 9.

Kubernetes provides all features to implement a dynamic scalable service infrastructure such as, workflows for rolling out services, replica management, or secret and configuration management. Kubernetes is on top of the container runtime such as Docker, and provides orchestration tooling, necessary to run large scale containerized service infrastructures. Nevertheless, sometimes a pure container orchestration platform like Kubernetes is not enough, which for instance provides no features for setting up a complete release workflow. This can be overcome with the usage of PaaS platforms like Openshift, which is discussed in the following chapter.

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 16. 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 expected from a PaaS platform[Ama18b; Goo18a; Mic18a; Ope18a].

PaaS providers usually provide templates for the major programming languages, application servers, as well as integrations to other cloud services. 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 customers to use their services over the services of the competition. What all PaaS providers have in common, is the consumption based pricing model, whereby only the consumed physical resources have to be paid for.

IPaaS can be seen as an enhancement of PaaS, which provides special features 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 services effortlessly, via a low/no code platform, whereby service integrations can be defined via an UI, rather then by implementing source code. Red Hat 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 Red Hat Openshift Enterprise. 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 previously noted ones, which can be self hosted or hosted by a local provider. The other major PaaS providers such as Microsoft Azure are only available as a cloud service, hosted in the vendors data centers[Ope18e].

5.1 The need for Platform as a Service

PaaS platforms like Openshift are based on a CaaS platform, and enhance CaaS by providing a web console and templates, which allow to define all necessary resources for a service, as well as its release behavior. An instance of a service can be created via the web console, by providing the required template parameter values. CaaS alone is suitable if its used by developers, but is not suitable for persons without any deep knowledge of containerization and container orchestration. This is where PaaS platforms come into place. Developers specify templates for the provided services, which contain all technical parts of a service infrastructure, and non-developers create a service instance via the web console. The PaaS platform takes the template and provided template parameter values and provisions the service for the user.

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 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 platform. The self service console could be implemented by enterprises for their specific use cases, whereby the custom self service console interacts with PaaS platform via its exposed REST 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 on the PaaS platform in a predefined manner. 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 provides additional automation.

5.2 Openshift

Openshift is a open source PaaS platform, which uses Docker as the container runtime and Kubernetes for the container orchestration. Openshift is designed as a client server architecture and a master slave architecture, the same way as Kubernetes, which has been discussed in Section 4.2 on page 17. 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 next page. Openshift introduces the concept of a project, which is actually a Kubernetes Namespace, whereby a project is completely isolated from other projects, as discussed in Section 5.2.2 on the following page. The following Figure 5.1 on the next page illustrates the architecture of an Openshift Cluster[Ope14; Ope18f].

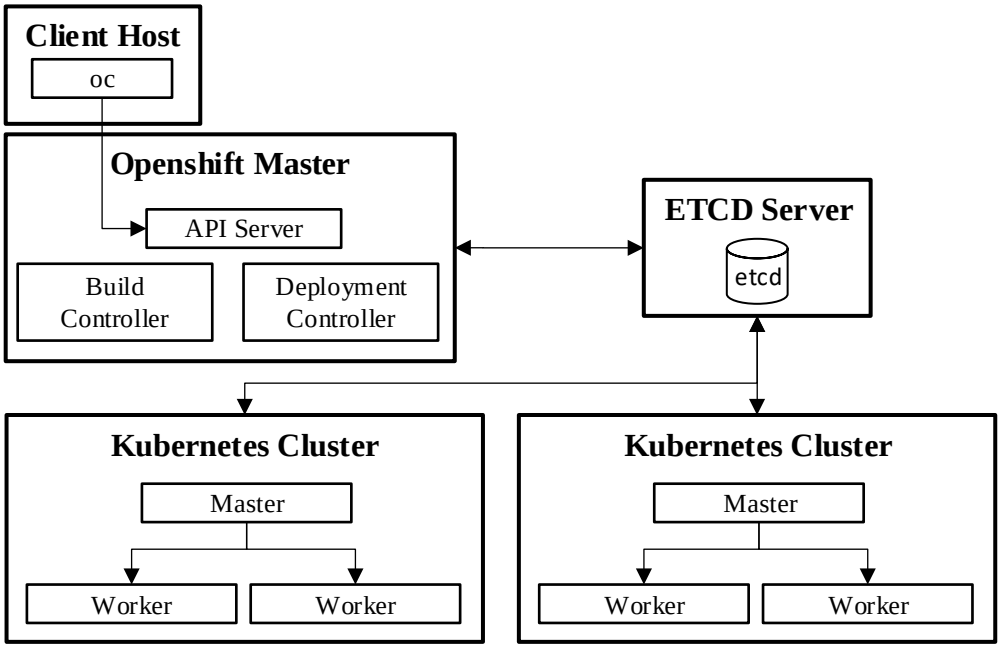

Figure 5.1: Architecture of an Openshift Cluster

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. Openshift is placed on top of Kubernetes, whereby 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, which are not provided by Kubernetes, 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 for Kubernetes Namespaces, which are not provided by Kubernetes itself. The Figure 5.2 on the following page illustrates the Openshift Project architecture, its contained Objects, and their dependencies to each other. The bold marked objects represent the Openshift Objects, which were introduced by Openshift.

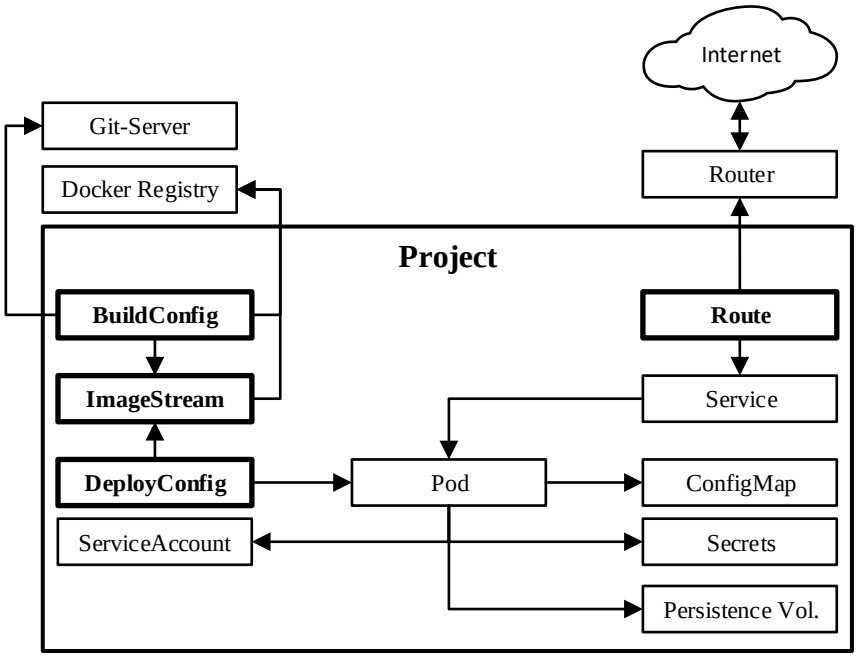

Figure 5.2: Architecture of a Openshift Project

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 17. The following sections briefly introduce the new objects provided by Openshift.

Openshift BuildConfig

An Openshift BuildConfig specifies the way how a Docker Image is built on the Openshift Cluster, whereby the built Docker Image is pushed into the Openshift internal Docker Registry. An Openshift BuildConfig supports the following listed strategies:

- The *Source-to-Image (S2I)* strategy, is the build strategy, which builds a Docker Image from a Dockerfile and application 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 builds 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 sources for a build are provided via a git repository, and an Openshift BuildConfig can be triggered via a web hook by an external git server.[Ope18b; Ope18g].

Openshift ImageStream

An Openshift 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*), whereby

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

An Image Stream-Tag references a Docker Image instance by its Docker Image-Id, because a Docker Image-Id always references the same Docker Image instance, whereby a Docker Image-Tag can reference multiple Docker Image instances. Once the Docker Image has been imported, it will not be automatically pulled again, unless the Image Stream-Tag is *latest*.

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 avoids this by referencing the Docker Image-Id, which makes a used Docker Image immutable within an Openshift Project, unless the latest version is explicitly defined[Ope18c].

Openshift DeployConfig

An Openshift DeployConfig specifies how a deployment of a Pod has to be performed. An Openshift DeployConfig 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). An Openshift DeployConfig supports the following listed deployment strategies[Ope18d]:

- 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

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

The following section will discuss the Enterprise Service Bus, which will be implemented by the prototype and running on an Openshift Cluster.

Chapter 6

Enterprise Service Bus

An Enterprise Service Bus (ESB) is a architectural pattern of the EAI patterns, which describes a distributed computing architecture, whereby 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 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 or internal services. JBoss Fuse is based on JBoss EAP, and bundles common frameworks used for integrations such as Camel, and is responsible for orchestration and mediation of the services. Camel provides API and implementations for the EAI patterns and is widely used in the industry, when it comes to integrate services[IA18; The15].

The integration of internal as well as external services has become more important over time, especially with the appearance of cloud platforms like PaaS. Common ESB middleware on the market usually define the ESB as a single application, which contains all integration services, whereby all integration services share the same life-cycle. A single application representing an ESB, is contrary to its definition as a distributed architectural model, but has the advantage, that the ESB is compiled at once, and therefore its contained services have to be consistent. Over time, the handling of the single ESB application will become inconvenient and complex, which is a common problem for a monolithic application.

The increasing 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 well 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, whereby the integrations are 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].

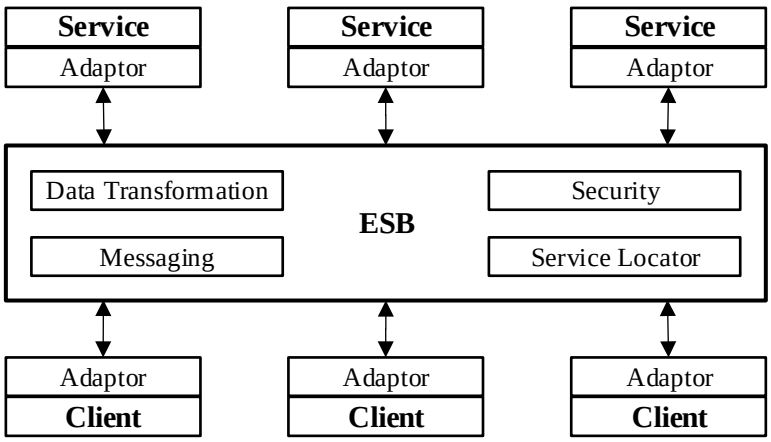

Figure 6.1: Architecture of an ESB

Figure 6.2 on the next page illustrates a bi-directional integration of services between two partner enterprises, whereby each integrated service is allowed to be consumed by the partner's customers, but only if the service is accessed via the partner's infrastructure. The ESB of the enterprises integrate the partner's provided services 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.

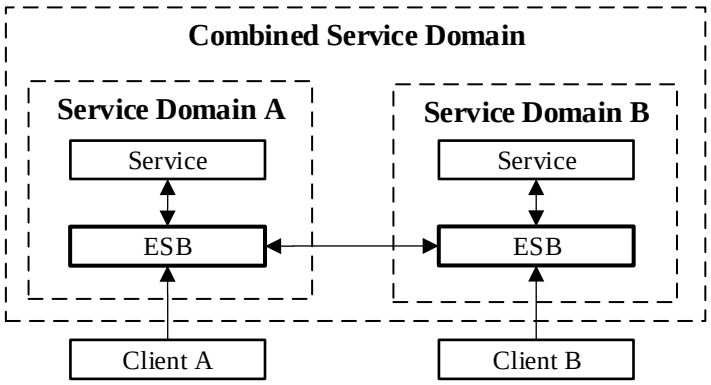

Figure 6.2: Architecture of a bi-directional enterprise integration

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 implement services under the consideration of the EAI patterns. Before the appearance of cloud platforms like PaaS, ESB implementations used existing platforms such as JBoss EAP, OSGI or Karaf for the service orchestration and mediation. In case of JBoss EAP, the platform provides all libraries and frameworks developers need for implementing a service. Commonly, the services are managed within a single application, which represents the ESB application. This is a monolithic approach of organizing services, but has the advantage that the management of the services is easier, because the source code is not separated. Figure 6.3 illustrates the monolithic organization of the services within a single ESB application, which is hosted on JBoss EAP.

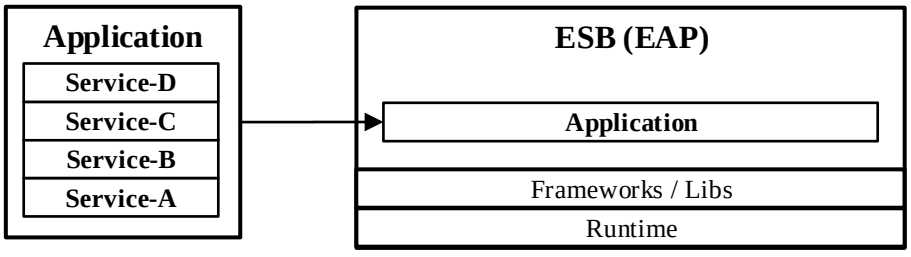

Figure 6.3: ESB running on EAP

Figure 6.3 was called an ESB, but is actually not one, because no services are distributed. Nevertheless, the industry calls such installations an ESB. The services of

such an ESB application are implemented separately, but still managed within a single code base, and communicate with each other, like they where on a bus system. JBoss EAP proxies the requests between the services, and therefore acts more like a message broker of an Hub and Spoke architecture, rather than acting as a bus system. The Hub and Spoke architecture was introduced, due the fact, that the count of edges of a fully connected graph requires $n/2 * (n - 1)$ edges, whereby the count of edges grows with the square of the count of nodes. A node in the graph represents a service, and an edge in the graph represents the connection between two services[Hoh03; HW08].

With the appearance of cloud platforms such as PaaS, the PaaS platform can now take over the mediation, and security aspects of an ESB. The services can be completely separated and de-coupled from each other, designed as microservices with their own life-cycle, and be managed by the cloud platform. Additionally, the services are hosted in a clustered infrastructure, 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 additional tooling for implementing services[Liu+15a; Liu+15b].

6.4 ESB with Openshift

With the appearance and general availability of cloud platforms like PaaS, it is now possible to move an ESB into the cloud, whereby the cloud platform takes over some aspects of the ESB middleware such as mediation and security. Openshift performs mediation via the Openshift Service abstraction, whereby multiple service instances can be present behind the Openshift Service, and the security is applied by isolating the Openshift Projects, and making services only accessible within an Openshift Project, or via an Openshift Route. The service orchestration is not needed anymore, because the services are now microservices, which need service choreography. The main difference between service orchestration and service choreography is, that the service orchestration has a local point of view and a central orchestrator component, whereby the service choreography has a global point of view and no central orchestrator component[Ric15].

A main problem of existing monolithic ESB applications is the fact, that all services are managed within a single application, with one release version, and a shared life-cycle. If the ESB is represented by a cloud platform, the services have to be implemented as separated services, which forces developers to separate their services into separate code bases, and to provide a proper designed and managed public API for their services. Figure 6.4 on the following page illustrates the services of an ESB application, which runs on Openshift.

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 become more flexible.

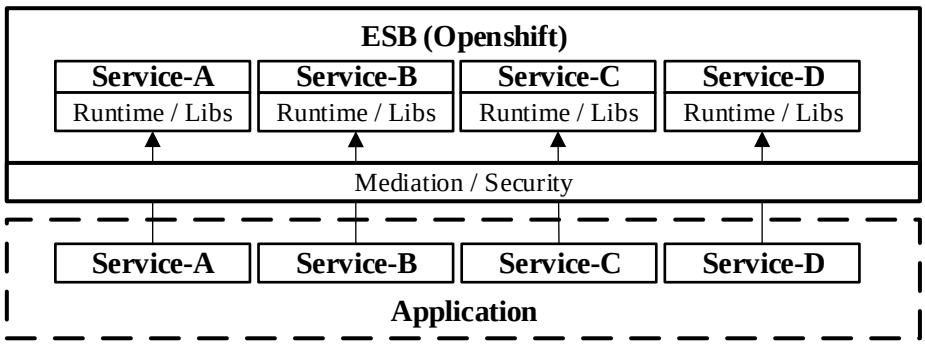

Figure 6.4: ESB application architecture with microservices

6.5 Integration example

This section will discuss an integration example, and how it would have been designed as part of a monolithic ESB application with JBoss Fuse based on JBoss EAP. The integration discussed in this chapter, will be the base for the prototype of this thesis, which is specified in Chapter 7 on page 32. Figure 6.5 illustrates the integration example, 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.

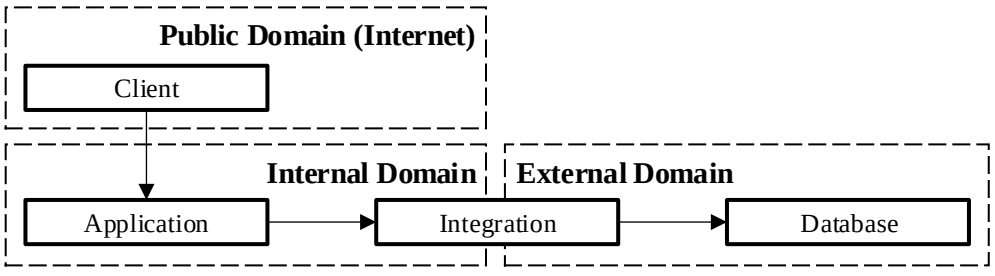

Figure 6.5: Integration Service-Domains

Figure 6.6 on the next page illustrates the design of the integration in an monolithic ESB application with JBoss Fuse based on JBoss EAP and Switchyard, which provides an API for implementing a 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)[Red18f].

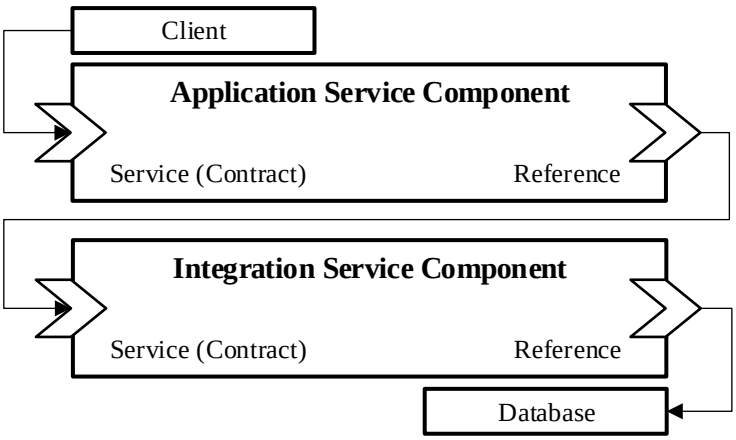

Figure 6.6: Integration Services with SCA

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, and security is performed by the ESB middleware. Additionally, the ESB middleware provides bindings for commonly used technologies such as REST or SOAP, which can be applied to 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 following chapter will specify the prototype based on the introduced integration example, and will show that an ESB can be implemented on Openshift.

Chapter 7

Design ESB in Openshift

In this chapter, the ESB integration example of Section 6.5 on page 30 will be designed as microservices, which can run on a Openshift Cluster. An Openshift Project will act as the ESB, which will provide the mediation, security, configurations and secrets for the services. The concrete functionality of the services is considered to be not important. The goal is to redesign the service components of Figure 6.6 on the previous page as microservices, and to design the Openshift Project, which will host the services, and act as the ESB.

As discussed in Chapter 6 on page 26, an ESB is a distributed computing architecture, whereby services act as a consumer or producer. These 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 the ESB middleware. The prototype will illustrate that SOA service components can be implemented as microservices, whereby features provided by the ESB middleware, such as 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 example of Figure 6.6 on the preceding page. 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 an ESB middleware, a microservice is completely separated from the other services, brings in its own libraries, and runs in its own runtime environment. From the implementation point of view, the microservices need to provide an runtime environment, which was previously 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 JBoss EAP, every service component is accessed via a reference, whereby the reference proxies the request directly to the service, which is running in 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.

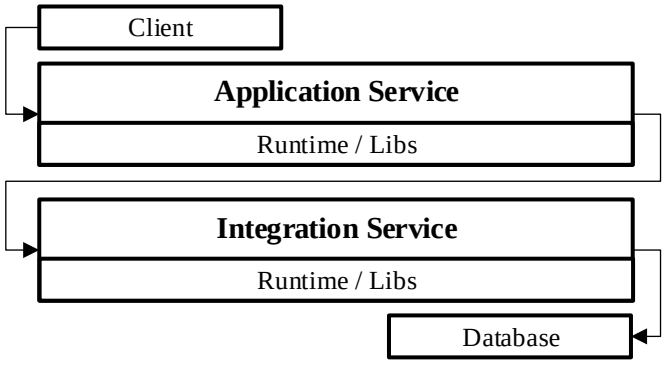

Figure 7.1: Microservice architecture

7.2 Microservice Aspects

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

Figure 7.2 on the next page illustrates the hierarchy of the microservice aspects, which will ensure that the implemented services are

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

The prototype will be implemented in Java, whereby the Eclipse community has provided the MicroProfile specifications, which provide an API to cover most of the microservice aspects. Using the MicroProfile specifications decreases the implementation effort, and provides an standardized way to implement microservices[Ecl18b].

The following sections will specify the microservice aspects for the to implement microservices.

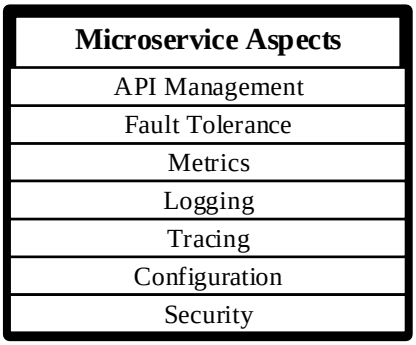

Figure 7.2: Service Requirements

7.2.1 Security

Distributed services need to be secured properly from unauthorized access, therefore the microservices will be secured with OAuth2. OAuth2 is a token based authentication scheme, which has become popular over past few years. There are several open source implementations for interacting with an authentication service via the OAuth2 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, and the Integration Service validates the provided token against the central authentication service[oau18].

7.2.2 Configuration

The Eclipse MicroProfile specifications provide the MicroProfile-Config specification, which provides an API to inject configuration parameters into CDI Beans. 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 services must be implemented in a way, to be configurable for different stages such as DEV (development) and PROD (production), whereby the services are only allowed to use configuration parameters via injection[Ecl18a].

7.2.3 Tracing

The Eclipse MicroProfile specifications provide the MicroProfile-OpenTracing specification, which provides an API for tracing an application on a method level and 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, whereby 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, provided by the distributed services. The services must provide all of their logging to a central service, whereby the logs must be marked with a transaction id, which is provided by the MicroProfile-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 Eclipse MicroProfile specifications provide the MicroProfile-Metric specification, which provides an API to define and manage metrics. Metrics allow to comprehend the state of a microservice, such as resource consumption, API calls or failure counts. Metrics along with distributed tracing and distributed logging, provide the necessary data, developers and operators need to analyze failures in services, which occurred in service call chains. The services must collect reasonable metrics and publish those metrics, which can be made available to a central metric service[Ecl18d].

7.2.6 Fault Tolerance

The Eclipse MicroProfile specifications provide the MicroProfile-FaultTolerance specification, 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[Ecl18c].

7.2.7 API Management

The API management of a public API such as REST API and REST Models ensure, that the clients, using the 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 an API, whereby 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. The services must provide a Swagger Documentation, and provide a capability to access the Swagger Documentation[Par18; Sma18].

7.3 Openshift Architecture

Figure 7.3 illustrates the design of the Openshift Project, which will act as the ESB. As discussed in Section 5.2.2 on page 23, Openshift isolates Kubernetes Namespaces by bringing in the concept of an Openshift Project. Services in an Openshift Project, which are not exposed via an Openshift Route, are implicitly protected from access from external networks, and 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.

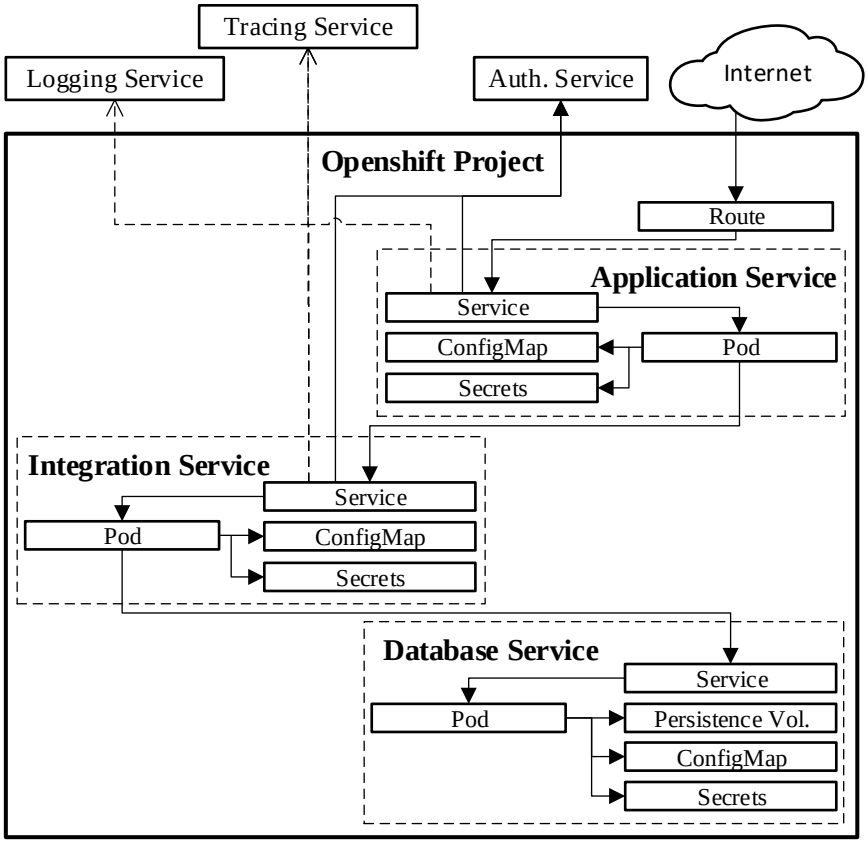

Figure 7.3: Openshift Project architecture

All services provide their tracing and logging data to external services, whereby no special configuration will be needed in Openshift, because Openshift allows services to access external resources. The mediation of the services is managed by Openshift Service objects, whereby the services communicate with each other via their service names, which ensures, that the communication stays within the Openshift Project. The Openshift Project must manage all configurations and secrets, which can be referenced by the services.

7.4 Openshift Requirements

The implementation of the Openshift Project such as templates and scripts, must be implemented under consideration of the principles of IaC, which have been discussed in Section 2.2 on page 6. Sticking to the principles of IaC will ensure, that the Openshift Project can effortlessly be managed and reproduced on any Openshift Cluster.

The Openshift Project will host the services and manage their configurations and secrets via Openshift ConfigMaps and Openshift Secrets, which have been discussed in Section 4.2.1 on page 17. The configurations and secrets are managed outside the service code bases, and are provided by the 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 shouldn't be accessible by developers.

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

For demonstration purpose, the Openshift Project, representing the ESB, will host a tracing service and a log aggregation service as well. This is due to the fact, that in shared environments, the customers have limited privileges, and therefore, not all configurations of Openshift are available.

The following chapter will discuss the implementation of the 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 lots of Java Enterprise Frameworks and Java Enterprise-Platform specifications, which are beyond the scope of this thesis, therefore a focus will be set on the implementations of the aspects discussed in Section 7.2 on page 33. Is is assumed, that the reader is familiar with Java Enterprise Development, Java Enterprise Frameworks, Maven and the microservices architecture. The implemented prototype is available on Github [1]. The repository contains a *README.adoc* file, which describes how to setup the prototype on a Windows Host.

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

As the prototype illustrates, the ESB is represented by an Openshift Project on an Openshift Cluster, whereby the Openshift Cluster acts as the platform for the hosted services, and the Openshift Project represents the ESB. The services will be hosted in an Openshift Project, whereby the Openshift Project provides features as discussed in Section 5.2.2 on page 23 for managing the life-cycle of the hosted services. The implemented resources for managing the Openshift Project are discussed in Section 8.8 on page 55.

The following Section 8.1 on the following page will briefly introduce the used technologies and frameworks, which were used for implementing the services with the microservice architecture.

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

8.1 Microservice Technologies

The following sections will give a brief introduction about the most important technologies and frameworks, used to implement the services as microservices. Each implemented service is setup the same way, because the concrete purpose of the service doesn't matter, when the microservices have to be integrated into a distributed service network. All of the following technologies and frameworks provide all necessary API and implementations for implementing a microservice, which is hosted on an Openshift Cluster.

8.1.1 JBoss Fuse Integration Services 2.0

JBoss Fuse Integration Services 2.0 is a set of tooling for developing services running on an 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 services during runtime. Additionally, a Maven Plugin is provided, which allows to interact with the Openshift Cluster during a Maven build, whereby the service life-cycle on an Openshift Cluster can be managed via Maven Goals. JBoss Fuse Integration Services 2.0 allows developers to interact with an Openshift Cluster in a way, like developers did before with an application server like Wildfly[Hußß18; Pro18].

8.1.2 Wildfly Swarm

Wildly Swarm is the Java Enterprise 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 Uber-Jar is a repackaged Java Web-Archive, which could be hosted on any application server, which provides all of the referenced dependencies, which are added during the repackaging[Red18e].

During the implementation of the prototype, Wildfly Swarm has been renamed to Thorntail.io[2], whereby the Maven Group-Id has been renamed from *org.wildfly.swarm* to *io.thorntail*. The used Maven Dependencies are still the same, and can be easily migrated by renaming the Maven Group-Id. The implemented sources are not affected at all.

8.1.3 Fabric8

Fabric8 is an integrated development platform for developing applications on Kubernetes. Fabric8 provides a 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. With

[2] https://thorntail.io/

the Maven Plugin, the interaction of developers with Openshift is reduced to a minimum, for instance, no custom Openshift BuildConfig has to be provided, because it is generated by the Maven Plugin itself[Red18b].

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

8.2 Security

The services are secured with OAuth2, 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 needs to be added as a dependency to the Maven *pom.xml*. The adapter needs a configuration, to know what resources are secured.

8.2.1 Service

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

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

Listing 2: Keycloak-Adapter 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 Endpoints.

```
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 on the following page 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 is be replaced by Maven Properties, whereby the Maven Properties can be provided in the *pom.xml* or provided/overwritten by Java Options during the Maven Build process.

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

Listing 4: Secret configuration in the deployment.yml

Listing 5 show the specification of the mount of the Docker Volume, which provides the secrets for the application. 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 following page.

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

Listing 5: Configuration volume mount in deployment.yml

8.2.2 Openshift

This section will discuss the Openshift implementation, whereby the implementation is represented by a shell script, which manages the Openshift Secrets. The secrets are managed outside the code bases of the services, and are supposed to be maintained by operators.

Listing 6 on the next page shows the Openshift CLI-Commands, which are used to create the Openshift Secrets. The first command creates an Openshift Secret from literal values, which provides the configurations for the service. The second command creates an Openshift Secret from a file, whereby the filename is the secret key and the file content is the secret value. The secret file is used by the Keycloak Adapter to validate the client OAuth tokens.

```
oc create secret generic "${SECRET_SERVIVE}" \
   --from-literal="service.db.base-url=${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

This section discussed the implementations, which are necessary to secure services hosted on an Openshift Cluster via OAuth2 with Keycloak. No source code is necessary, only configuration. The following Section 8.3 will discuss the configuration of the 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 services use the MicroProfile-Config specification, to be configurable for multiple stages. The services consume the configuration parameters via injection, which can be provided from different configuration sources. Developers are bound to the configuration/secret name, keys and their value type. Developers are not bound to the configuration/secret source, which allows to provide configurations/secrets from different sources and for different stages.

8.3.1 Service

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 implementation, to enable injectable configurations.

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

Listing 7: MicroProfile-Config dependency in pom.xml

Listing 8 on the following page shows the definition of the configuration source in the *project-stages.yml* for the development stage, whereby the configuration parameter values 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 in the *project-stages.yml* for the production stage, whereby the configuration parameter values are loaded form files, located in the configured directory. The directory location is represented by a Maven Property, because its used in multiple configuration files, as already discussed in Section 8.2.1 on page 40.

```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 source of the configurations is unknown. The injected configuration properties are retrieved from an Openshift Secret, but are used in the source code the same way as configurations.

```java
@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

The Openshift implementation has already been covered by Section 8.2.2 on page 41, because all of the configurations are managed as Openshift Secrets, because they contain sensitive data.

8.4 Tracing

The services use the MicroProfile-OpenTracing specification to provide tracing data to a central tracing services. Jaeger[3] is used as the tracing service, which collects all tracing data and provides a GUI for analyzing the collected traces.

8.4.1 Service

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

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

Listing 11: MicroProfile-OpenTracing dependency in pom.xml

Listing 12 shows the configuration in the *project-stages.yml*, for the integration into the Jaeger tracing service, whereby the configuration parameters are provided by Maven Properties, environment Variables and literals. 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 Jaeger in project-stages.yml

Listing 13 on the next page 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 CDI Interceptor, which implements the tracing logic.

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 the call duration. The CDI Interceptor

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

creates a new span for each method invocation, and appends the created span to an existing parent span, or the created span is the parent span.

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

Listing 13: Enable tracing for a CDI Bean

8.4.2 Openshift

The communication between the services and tracing service is done via UDP protocol, and therefore Openshift does need any special configuration. Openshift doesn't interfere with outgoing connections, only incoming.

8.5 Logging

The services provide logging to a central log aggregation service. Graylog[4] is used as the log aggregation service, which collects all logging data and provides a GUI for analyzing the aggregated logs.

8.5.1 Service

This section will discuss the implementation of the service logging. Listing 14 shows the dependencies, which bring in the logging implementations. SLF4J[5] has been chosen as the logging facade, whereby an integration into 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: ogging dependencies in pom.xml

The following listings are part of the *project-stages.yml* configuration file and configure logging for different stages. Listing 15 on the next page shows the configuration of the logging format, which uses Mapped-Diagnostic-Context (MDC) parameters

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

to mark a log entry with the transaction id. The configured formatter is used for all stages, because it has been defined globally.

```
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: Logging format configuration in project-stages.yml

Listing 16 shows the logging configuration for the development stage.

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

Listing 16: Logging configuration for development stage in project-stages.yml

Listing 17 on the next page shows the configuration in the *project-stages.yml* for the logging of the production stage, whereby logs are send to a central log aggregation service. A Syslog Log-Handler is configured, which sends the logs to the log aggregation service via the Syslog[6] protocol. The configuration, where to send the logs, is provided via System Properties, which are created during startup out of environment variables, which are set with values of an Openshift Secret.

Listing 18 on the following page shows the implementation of the interface *ContainerRequestFilter*, provided by the JAX-RS specification, which is used to capture the current trace transaction id on the REST Endpoints. The implementation is depending on the Uber MicroProfile-OpenTracing implementation, because the specification itself doesn't provide an accessor for the transaction id yet. The captured transaction id is set into MDC, whereby the formatter, defined in Listing 15, references the MDC parameter.

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

```
GRAYLOG_HOST: "${env.GRAYLOG_HOST}"
GRAYLOG_PORT: "${env.GRAYLOG_PORT}"
GRAYLOG_PROT: "${env.GRAYLOG_PROT}"

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

Listing 17: Configuration of the logging for production stage

```
@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

Listing 19 on the following page shows the CDI Producer method, which provides the logger instances for injection points. The logger is produced for the Dependent Scope, which means, that the life-cycle of the logger instance is managed by the CDI Bean, which gets the logger injected.

```
@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: CDI Producer for dependent scoped logger instances

Listing 20 shows a class using an injected logger to log a info message. As this examples illustrates, the user code has no knowledge about a log aggregation backend or about a transaction id.

```
@ApplicationScoped
public class ReportServiceImpl implements ReportService {

    @Inject
    private Logger log;

    @Override
    public ReportModel generateReportForCustomer(Long id) {
        log.info("Generating report for customer id");
        ...
    }
}
```

Listing 20: Logger usage

8.5.2 Openshift

The services send their logs to a central log aggregation service via the UDP protocol, and therefore no special settings for Openshift necessary. Nevertheless, logs send to the console are collected by Openshift and can be analyzed in the Openshift Web-Console.

8.6 Fault Tolerance

The services use the MicroProfile-FaultTolerance specification to define fault tolerance behavior on methods. Hystrix[7] is a popular framework for failure handling in applications, was the inspiration for the MicroProfile-FaultTolerance specification, and is used as the back-end for the Wildfly Swarm provided dependency.

8.6.1 Service

This section will discuss the implementation of the service fault tolerance. The services use the MicroProfile-FaultTolerance specification to define the service fault tolerance behavior, which defines the service resilience. Listing 21 shows the dependency, which brings in the fault tolerance API and implementation.

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>microprofile-fault-tolerance</artifactId>
</dependency>
```

Listing 21: MicroProfile-FaultTolerance dependency in pom.xml

Listing 22 shows the CDI Producer method for producing the Keycloak token, which defines fault tolerance behavior. The special use of the method as a CDI Producer doesn't affect the fault tolerance logic, and can be applied on any CDI Bean. Each time when the producer method is called, a token request is send to Keycloak, to retrieve an access token. The invocation is retried 5 times with a 100 millisecond delay, and each invocation is timed out after 5 seconds.

```
@Produces   // CDI Producer
@OAuthToken // CDI Qualifier
@Dependent  // CDI Scope
// Retries 5 times with 100ms delay on given exceptions
@Retry(delay = 100L, maxRetries = 5, retryOn = {
  TokenResponseException.class})
// Timeout invocation after 5 seconds
@Timeout(value = 5L, unit = ChronoUnit.SECONDS)
String obtainOauthToken() throws IOException {
    return tokenRequest.execute().getAccessToken();
}
```

Listing 22: Fault tolerance definition on CDI Producer method

8.6.2 Openshift

The fault tolerance behavior as discussed in Section 8.6.1 only affects the service itself and not Openshift. But, Openshift provides a kind of fault tolerance, for instance,

[7] https://github.com/Netflix/Hystrix/wiki/How-it-Works

by restarting crashed Pods, and therefore ensuring, that the defined replica count of service is met by the Openshift Cluster.

8.7 REST API-Management

The services use Swagger[8] to provide documentation for their REST API. Swagger provides an intermediate format, which can be used by tooling, for testing and client generation. The REST API represents the public view of the service, which is implemented in a way, so that it is de-coupled from the service logic, and supports several ways to perform API migrations.

The following sections will discuss the implementation of the REST API-Management on the service side, and the REST API usage on the client side. Both use Swagger, whereby the service provides Swagger Documentation, and the clients are generated by using the service provided Swagger Documentation.

8.7.1 Service

This section will discuss the management and implementation of the service REST API. Listing 23 shows the service dependencies, which bring in the

- the Java BeanValidation implementation,
- the JAX-RS Server implementation,
- the JAX-RS Server Java-BeanValidation integration,
- and the Swagger API and implementation.

```
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>bean-validation</artifactId>
</dependency>
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>jaxrs</artifactId>
</dependency>
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>jaxrs-validator</artifactId>
</dependency>
<dependency>
  <groupId>org.wildfly.swarm</groupId>
  <artifactId>swagger</artifactId>
</dependency>
```

Listing 23: JAX-RS/BeanValidation/Swagger dependencies in pom.xml

Listing 24 on the following page shows the *swarm.swagger.conf* configuration file, which configures Swagger for the documented service. During startup, Swagger will

[8] https://swagger.io/

scan the configured packages for interfaces and classes, which provide documentation in form of Swagger Annotations. The scanned documentations are written to a file named *swagger.json*, which contains the Swagger Documentation of the service REST API.

```
packages: "com.gepardec.esb.prototype.services.app.configuration,
           com.gepardec.esb.prototype.services.app.rest"
root: "/rest-api"
version: "1"
title: "${project.artifactId}"
```

Listing 24: Swagger configuration in swarm.swagger.conf

Listing 25 shows an interface, which specifies a REST Endpoint via JAX-RS Annotations, and provides documentation via Swagger Annotations. Additionally, Java BeanValidation-Annotations are used to define constraints for the input arguments of the REST Operations, so that validation is applied on all incoming requests. The JAX-RS, Java BeanValidation and Swagger Annotations are scanned and applied to the generated Swagger Documentation.

```
@Path("/report")
@Api(value = "ReportRestService",
     description = "The api for generating reports for customers")
public interface ReportRestService {

    @Path("/generate")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @ApiOperation(value = "Generate report for the given customer id",
                  response = ReportModelDto.class)
    ReportModelDto generate(@QueryParam("id") @NotNull @Min(0) Long id);
}
```

Listing 25: JAX-RS interface with Swagger Annotations

8.7.2 Client

This section will discuss the implementation of the REST Client, which is generated by the swagger Maven-Plugin by using the service provided *swagger.json* file. The following listings show the configuration of the Maven Plugins, which are used to generate a REST Client from a Swagger Documentation during a Maven Build.

Listing 26 on the following page shows the configuration of the Maven Helper-Plugin in the consumer service *pom.xml*, which is used to add the generated REST Client sources for the compilation. The source directory points to the directory, where the generated REST Client sources are located.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <version>${build-helper-maven-plugin.version}</version>
  <executions>
    <execution>
      <id>add-source</id>
      <phase>generate-sources</phase>
      <goals><goal>add-source</goal></goals>
      <configuration>
        <sources>
          <source>${generatedClientDir}/src</source>
        </sources>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Listing 26: Maven Helper-Plugin configuration in pom.xml

Listing 27 shows the configuration of the Maven Clean-Plugin in the consumer service *pom.xml*, which is used to clean the unwanted generated sources and resources. The Swagger Maven-Plugin generates a standalone REST Client, which cannot be turned off, but only the plain generated models and interfaces are needed.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-clean-plugin</artifactId>
  <version>${maven-clean-plugin.version}</version>
  <executions>
    <execution>
      <id>clean-swagger-sources</id>
      <phase>generate-sources</phase>
      <goals><goal>clean</goal></goals>
      <configuration>
        <!-- Only plain models and JAX-RS interfaces wanted -->
        <directory>${generatedClientDir}/src/main/java</directory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Listing 27: Maven Clean-Plugin configuration in pom.xml

Listing 28 on the following page shows the Swagger Maven-Plugin configuration in the consumer service *pom.xml*, which is used to generate the REST Client during the Maven Build. Custom Swagger Code-Generator templates are used, due to the fact that, there is no Swagger Code-Generator, which only generates plain JAX-RS interfaces and models.

```
<plugin>
  <groupId>io.swagger</groupId>
  <artifactId>swagger-codegen-maven-plugin</artifactId>
  <version>${swagger-maven-plugin.version}</version>
  <executions>
    <execution>
      <id>generate-swagger-integration-database</id>
      <goals><goal>generate</goal></goals>
      <configuration>
        <language>jaxrs-cxf</language>
        <inputSpec>${apiDir}/app.json</inputSpec>
        <output>${generatedClientDir}</output>
        <apiPackage>${apiPackage}</apiPackage>
        <modelPackage>${modelPackage}</modelPackage>
        <templateDirectory>${templateDir}</templateDirectory>
        <generateSupportingFiles>false</generateSupportingFiles>
        <addCompileSourceRoot>false</addCompileSourceRoot>
        <modelNamePrefix>App</modelNamePrefix>
        <configOptions>
          <sourceFolder>/src</sourceFolder>
          <dateLibrary>java8</dateLibrary>
        </configOptions>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Listing 28: Swagger Maven-Plugin configuration in pom.xml

Listing 29 shows the generated JAX-RS interface, which was generated out of the the Swagger Documentation, which was provided by the generated *swagger.json* file. The generated JAX-RS interface is very similar to the original JAX-RS interface, but could contain differences, for instance, when a REST Operation parameter of type string was documented as value type number.

```
@Path("/")
public interface ReportRestServiceApi  {

    @GET
    @Path("/report/generate")
    @Produces({ "application/json" })
   AppReportModel generate(@QueryParam("id") @NotNull @Min(0) Long id);
}
```

Listing 29: Generated JAX-RS interface

Listing 30 on the following page shows how to build an REST-Client for the generated JAX-RS interfaces, whereby developers work with the generated API, and the logic for handling the request and response is provided by RESTEasy[9]. Changes

[9] https://resteasy.github.io/

made to the Swagger Documentation could cause compile errors, because the signature of the generated interface could differ from the previous generated version. Therefore, the usage of the REST Client is type safe. The MicroProfile-OpenTracing specification provides a JAX-RS Client-Filter, which integrates the REST Client request into the configured tracing back-end. This integration ensures, that calls to another service, made via a REST Client, contribute to an existing trace, or are the start of a new trace. A custom JAX-RS Client-Filter has been implemented, which retrieves the OAuth2 Token from the CDI Container, and sets the token on the HTTP Authentication-Header of the client request.

```
Client client;
ReportRestServiceApi api;
ResteasyClientBuilder builder = new ResteasyClientBuilder();

// Configure and build client
client = builder.connectionCheckoutTimeout(2, TimeUnit.SECONDS)
                .establishConnectionTimeout(2, TimeUnit.SECONDS)
                .socketTimeout(2, TimeUnit.SECONDS)
                .connectionTTL(2, TimeUnit.SECONDS)
                .connectionPoolSize(200)
                // Appends Tracing feature for the JAX-RS client
                .register(ClientTracingFeature.class)
                // Sets OAuth token on Authentication header
                .register(AppendOAuthFilter.class)
                .build();

// Create proxy for JAX-RS interface
api = ProxyBuilder.builder(ReportRestServiceApi.class,
                           client.target(url))
                  // If rest operation consumes all types,
                  // then use 'text/plain'
                  .defaultConsumes(MediaType.TEXT_PLAIN)
                  .build();
```

Listing 30: Example of building a type safe REST Client

In the prototype, the built REST Clients are injectable into CDI Beans, whereby the Rest Clients are managed by a custom proxy, which applies proper fault tolerance behavior to the REST Client method calls.

8.7.3 Openshift

The REST API-Management and migration doesn't affect Openshift, because the services REST API is either accessible only within the Openshift Project network, or is exposed via a single Openshift Route. Openshift doesn't provide any mechanisms for redirecting requests to other services based on path conditions, as for instance Nginx is capable to do so.

8.8 Openshift Project

This section will discuss the implementation of the Openshift Project, which represents the ESB. The implementations are represented by scripts, configurations and secrets, which ensure that the Openshift Project is properly setup, and provides all resources needed by the services, such as Openshift ConfigMaps and Openshift Secrets. The Openshift resources, which are consumed by the services, are managed by one script per service.

The scripts, configurations and secrets are managed by operators, which ensure that the Openshift Projects are properly setup, and provide all Openshift resources for the services, for a specific stage, an Openshift Project represents.

8.8.1 Scripts

This section will discuss the implemented scripts for managing the Openshift resources consumed by the services. Listing 31 shows an excerpt of an implemented script, which manages Openshift Secrets for a service, which are created from files. The Openshift Secrets could also have been created from Openshift Templates, whereby the secrets are either hard-coded in the Openshift Templates, or are provided via Openshift Template-Parameters.

```
# For creating configmaps
# oc create configmap == oc create secret generic
function createSecrets() {
  oc create secret generic ${SECRET_SERVIVE} \
    --from-env-file=./config.properties

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

# For deleting configmaps
# oc delete configmap/config_name == oc delete secret/secret_name
function deleteSecrets() {
  oc delete secret/${SECRET_SERVIVE}
  oc delete secret/${SECRET_SERVIVE_KEYCLOAK}
}
```

Listing 31: Shell functions for managing Openshift Secrets via a CLI

The scripts are a convenient way for managing Openshift Secrets, and are more flexible then template based Openshift Secret-Management. With Openshift Templates, there would also be the need for scripts, which can create/modify/delete via Openshift Templates created resources. The scripts are separated from the actual secrets or configurations, and can therefore be used for all stages, whereby each stage provides their own secrets and configurations.

Listing 32 on the following page shows the CLI functions, for managing the database

application, which is used by the prototype, to simulate the external integrated database. The template encapsulates and defines all resources needed by this application, which are created during the instantiation process. This is a main feature of Openshift Templates, which allows to bundle all definitions into one template file.

```
# Function to create a database application via a template
function createService() {
  oc new-app -f ./postgres-full.json \
    -p "APP_NAME=${APP}" \
    -p "MEMORY_LIMIT=${MEM_LIM}" \
    -p "DATABASE_SERVICE_NAME=${DB_SERVICE}" \
    -p "POSTGRESQL_USER=${USER}" \
    -p "POSTGRESQL_PASSWORD=${PWD}" \
    -p "POSTGRESQL_DATABASE=${DB_NAME}" \
    -p "INIT_SECRET=${INIT_SECRET}" \
    -p "VOLUME_CAPACITY=${VOL_LIM}" \
    -p "POSTGRESQL_VERSION=${VERSION}"
}

# Delete the database application resources
function deleteService() {
  oc delete all -l app=${APP}
  oc delete pvc/${DB_SERVICE}
  oc delete secret/${DB_SERVICE}
}
```

Listing 32: Shell functions for managing an application via CLI

8.8.2 Templates

There is no need for additional Openshift Templates for the services, because Fuse Integration Services 2.0 and the Fabric8 Maven-Plugin provide all necessary resources and functionality to manage the services via their own code bases. The service code bases manage Openshift Templates, such as the *deployment.yml*, which defines the environment for a service, by specifying

- the injection of configurations and secrets,
- the health checks,
- the resource limits,
- the roll-out behavior,
- and the accessible ports.

The Fabric8 Maven-Plugin manages only the service resources of an Maven Project, by applying proper labels and accessing the resources by its labels.

The following Chapter 9 on the next page will evaluate and analyze the implemented prototype, if the prototype fulfills the specification of Chapter 7 on page 32, and if the prototype is a valid representation for an ESB in Openshift.

Chapter 9

Discussion ESB in Openshift

This chapter will discuss the implemented prototype of Chapter 8 on page 38, and will discuss some important tasks such as

- managing multiple environments,
- managing service security,
- managing multiple service versions,
- managing public API migration,
- and managing Adapters and Message Translators as services,

which are very important, when running an ESB on Openshift. Whenever possible, Openshift will be compared to JBoss EAP, which can be used as the platform for the ESB middleware JBoss Fuse, as discussed in Section 6.3 on page 28.

The prototype implementation represents an ESB running on Openshift, but is actually nothing more, than an application represented by several microservices, running on a PaaS platform. Instead of managing a monolithic ESB application as discussed in Section 6.3 on page 28, the ESB application is split into independent microservices, which are managed completely separately, as discussed in Section 6.4 on page 29. Because of the usage of Openshift as the platform, the implemented services of the prototype are distributed services per default, which is the nature of Kubernetes, which is the base for Openshift. Openshift takes care about the distribution of the Pods within the Openshift Cluster, and ensures, that the Pods are accessible via their related service abstraction.

When splitting any monolithic application into independent microservices, then the sum of resources needed by all microservices of the ESB, will be higher as the resources needed by a monolithic ESB. With microservices more resources are needed, due to the fact, that each microservice brings in its own libraries and runtime environment, whereby the monolithic ESB runs in one runtime environment, where all libraries are shared. Then more heterogeneous libraries are used by the microservices, then more resources they will need to run. This a trade off, which needs to be considered when designing microservices for an ESB or any other application, which consists of microservices, or when splitting up a monolithic application to a microservice architecture.

Resource	Monolith	1 Micros.	10 Micros.	20 Micros.
CPU/Cores	2	1	10	20
RAM/GB	3	0.5	5	10

Table 9.1: Resource comparison of monolith and microservices

Table 9.1 is not a real world example, but shows the trend of the resource need of the whole application, when a monolithic application is split up into microservices. When the application is split up into too many microservices, than the needed resources increase drastically.

9.1 Service mediation

The SOA Patterns specify a mediation layer, as layer which facilitates the communication across multiple services by providing an abstraction layer, which de-couples the services from each other. The mediation layer ensures, that services can seamlessly communicate with replaced or newly released services instances. Kubernetes provides the Kubernetes Services, which mediates the request among multiple service instances represented by the service Pods. The Kubernetes Service is a content unaware mediator, because the message remains unchanged, and only mediation between service instances is supported.

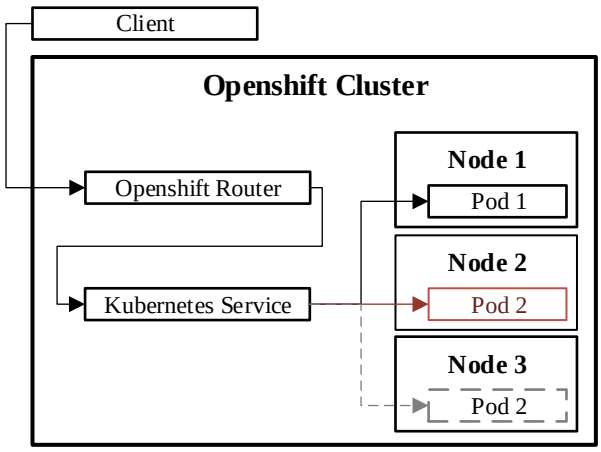

Figure 9.1: Openshift Service mediation

Figure 9.1 illustrates a scenario, where the service Pod 2 on Node 2 fails, and gets rescheduled on Node 3, whereby it gets a new IP address assigned. The Kubernetes Service is aware that Pod 2 has been rescheduled, and will balance request to the rescheduled Pod 2 located now on Node 3, when Pod 2 is fully up and running.

A Kubernetes Service performs a simple mediation by abstracting service Pods via service names, and load balancing of the requests to the service Pods depending on the chosen algorithm.

With an ESB, there is no central mediator needed, because an ESB is a distributed architectural model, which has no central component like a Hub and Spoke architecture. If message translation is needed, because of a consumer incapability of supporting the ESB used message format and protocols, then an adapter and message translator is needed as discussed in Section 9.6 on page 67.

The following sections will discuss the tasks introduced in the beginning of this chapter, and will show, that these task can be performed on the implemented prototype.

9.2 Managing Multiple Environments

An ESB is commonly hosted on multiple environments, whereby at least one productive and one testing environment should be present. These environments were commonly a VM, which provided the runtime environment for the ESB. As the prototype shows, the environment is now represented by an Openshift Project, which can be reproduced via scripts as discussed in Section 8.8 on page 55.

The services hosted on the ESB are using Fuse integration Service 2.0 and its provided tooling, which ensure, that the services are properly encapsulated in a container, and are properly managed in Openshift. Therefore, the service developers provide the necessary Openshift Templates, which has the effect, that the operators have no interaction with the service artifacts and service runtime environments anymore. Operators have to manage

- the Openshift Project, which contains the services,
- the Openshift ConfigMaps, which hold the service non-sensitive configuration,
- and the Openshift Secrets, which hold the sensitive service configuration.

Listing 9 on page 43 illustrates how developers reference Openshift Objects, such as Openshift ConfigMaps and Openshift Secrets, which are managed by operators.

Figure 9.2 on the following page illustrates the management and provisioning of multiple environments for an ESB, whereby the environment is represented by an Openshift Project. The Management Server pulls the scripts and Openshift Templates from a VCS Server, and the configurations from a Configuration Server, and uses them to provision new Openshift Projects, or manage existing ones. The scripts and Openshift Templates are separated from the configurations, which are actually providing the data for the scripts and Openshift Template-Parameters. With such an approach, the infrastructure becomes reproducible, versioned, and therefore consistent and disposable. These characteristics are also principles of IaC, which have been discussed in Chapter 2 on page 4.

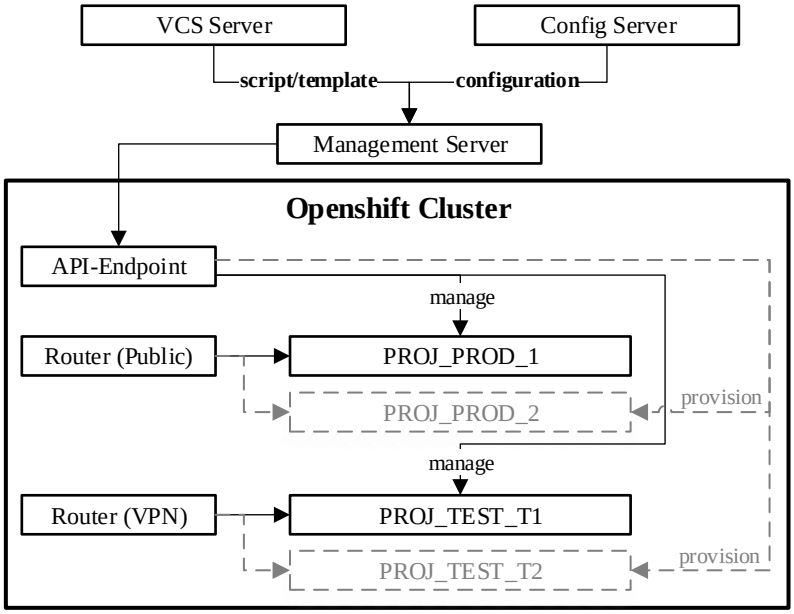

Figure 9.2: Management and provisioning of multiple environments

The interaction of the Management Server, VCS Server and Configuration Server, as illustrated in Figure 9.2, is similar to the Figure 2.2 on page 6, which illustrated, how a system can be reproduced with parametrized templates and an IaC tool. The Openshift CLI provides functionality to manage Openshift Objects, which is what needs to be done, when providing an environment in form of an Openshift Project, therefore the Openshift CLI acts as an IaC tool.

Table 9.2 on the following page compares the Openshift and JBoss EAP contained mechanisms, which provide the listed infrastructure features. As illustrated in the table, Openshift provides networking and isolation features, which are provided to JBoss EAP by its hosting environment, such as a VM. Except of the networking and isolation feature, JBoss EAP supports all other features either natively or by supporting a third party library. Nevertheless, Openshift combines all features in one platform, and makes them manageable via Openshift Templates and the Openshift CLI.

Openshift runs Docker Containers, and therefore the programming language, the service was implemented with doesn't matter, as long as the services can run in a container. Also, services hosted on PaaS platforms communicate via standard Protocols such as HTTP, which are commonly supported by almost any programming language. JBoss EAP on the contrary, runs only Java applications.

Feature	**Openshift**	**JBoss EAP**
Staging	Openshift Project	Server Instance
Management	Openshift CLI Openshift Web-Console Openshift REST-API	JBoss CLI JBoss Web-Console
Networking	Openshift Project Openshift Service Openshift Route Openshift Router	None (external)
Isolation	Openshift Project	None (external)
Configuration/Secrets	Openshift ConfigMaps Openshift Secrets	Java System-Properties Environment Variables Password Vault
Service Distribution	Openshift Worker-Node	Single JVM Karaf OSGI
Service Roll-out	Recreate Rolling	Framework dependent, normally recreate
Language Support	All, runnable in containers	Java

Table 9.2: Infrastructure feature comparison

The next section will discuss the service security within an Openshift Project, which can be managed the same way, as discussed in this section. Additionally to the security, provided by the isolation feature of an Openshift Project, the Management Server of Figure 9.2 on the previous page could also manage custom security configurations, which can be managed via the Openshift CLI as well.

9.3 Managing Service Security

With a common ESB middleware, the services are protected by running within a single runtime environment, or by security features provided by a supported third party library. In an Openshift Project, the services are implicitly protected by being isolated in a Kubernetes Namespace, as discussed in Section 5.2.2 on page 23, whereby the namespace of an Openshift Project cannot be accessed by other Openshift Projects without additional configuration. Services, which are supposed to be accessible from external networks, have to be explicitly exposed via an Openshift Route, which can handle secured connections, similar to an reverse proxy.

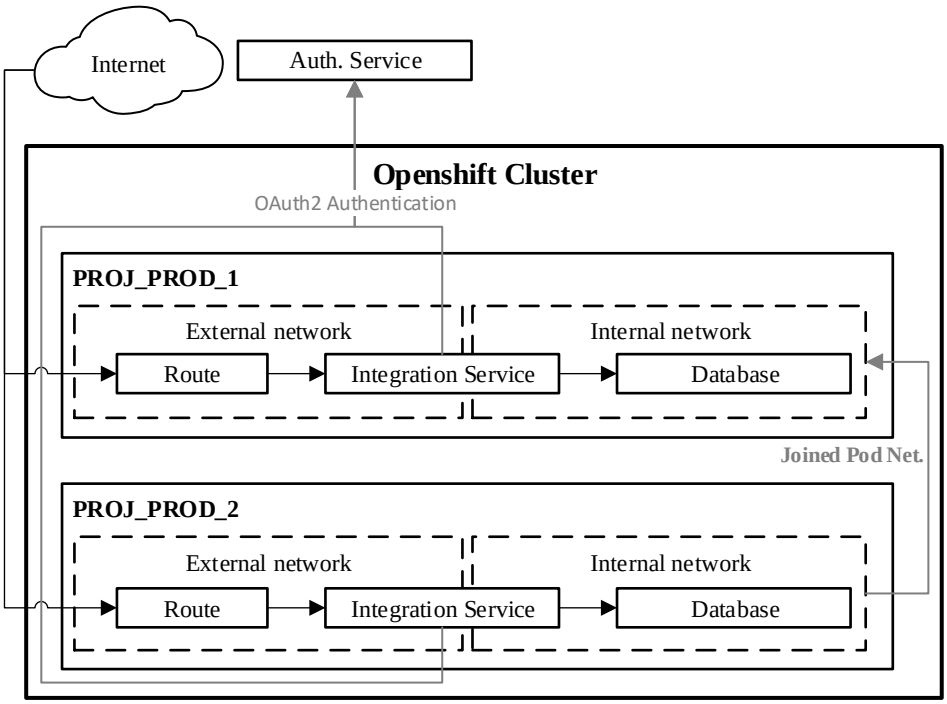

Figure 9.3: Service security in an Openshift Project

Figure 9.3 illustrates an example, similar to the implemented prototype, whereby two Openshift Projects host the same services, and where the Pod Networks of the two Openshift Projects are joined. The joined Pod Networks allow services hosted in project PROJ_PROD_2 to access services hosted in project PROJ_PROD_1 by their fully qualified service name. The fully qualified service name has the form of *<service>.<project_namespace>.svc.cluster.local*. Joining two Pod Networks is performed by Openshift Cluster-Administrators, and cannot be performed by developers.

Additionally to the isolation of the services within the Openshift Project, the Integration Service is secured via OAuth2, whereby the resource access is controlled by a central Authentication Server. On the one hand, the services are isolated within a Kubernetes Namespace, and on the other hand, additional security, such as access control, has to be provided by the service itself, because Openshift doesn't provide such a feature. It is not meant to isolate services from each other within an Openshift Project, because an Openshift Project, in particular a Kubernetes Namespace should only contain a set of services, which don't have to be isolated from each other.

Table 9.3 on the following page compares the Openshift and JBoss EAP contained

mechanisms, which provide the listed security features. As illustrated in the table, Openshift doesn't provide any support for access control on the service level, which is normal for a PaaS platform, such as Openshift. The services running in Docker Containers on an Openshift Cluster, have to implement access control, or have to use third party frameworks, such as Wildfly Swarm, which provide access control features as discussed in Section 8.2 on page 40. JBoss EAP on the other hand is a Java Application-Server, which provides support for access or user control for several providers.

Feature	**Openshift**	**JBoss EAP**
Network Isolation	Openshift Project	None (VM)
HTTPS	Openshift Router	Reverse Proxy
	Openshift Route	Endpoint Configuration
Access Control	None (external)	Endpoint Configuration
		Internal User-Database
		External User-Database
Single-Sign-On	None (external)	Endpoint Configuration
		Several SSO providers

Table 9.3: Security feature comparison

Openshift doesn't provide access control features to secure service resources, but provides security features such as user/group/role management, project permission management, Pod Network management, or quota management for Openshift Objects, such as Replication Controllers. Openshift uses Software-Defined-Networks (SDN), whereby the services don't have to handle communication security, because the Openshift Cluster will take care of it. Exposed services have an Openshift Route, whereby the Openshift Router, which acts as the reverse proxy, applies security to the communication on the external network. Within the isolated Pod Network, the services don't have to handle communication security, because this is handled by the Openshift Cluster.

9.4 Managing Multiple Service Versions

Sometimes it is necessary to run multiple versions of a service, for instance, if a new version is released, or if a consumer is not capable of migrating to the new version, but provides a significant business value for the enterprise. Openshift provides mechanisms to run multiple versions of a service in several ways. Figure 9.4 on the following page illustrates some scenarios for running multiple service versions on Openshift, whereby the illustrated scenarios of PROJ_1 and PROJ_2 are possible, because the old service version is N-1 compatible. N-1 compatibility means, that the service in the old version is capable of reading data written by the service in the new versions.

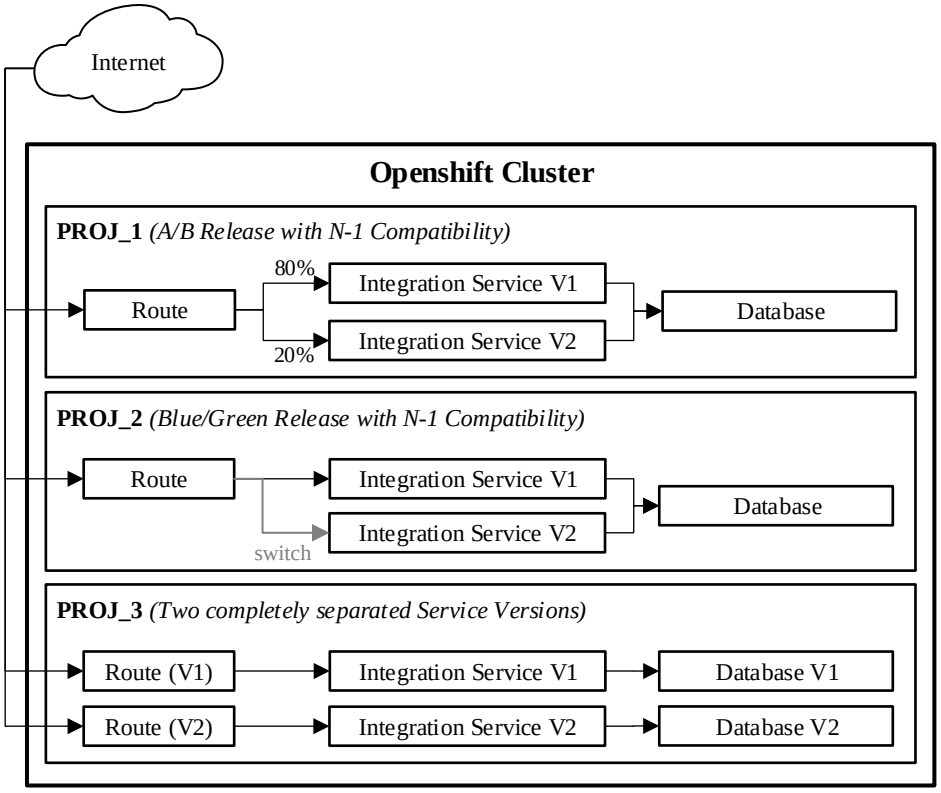

Figure 9.4: Running multiple service versions on Openshift

PROJ_1 of Figure 9.4 illustrates an A/B Release, which is used to run the old and new version of a service in parallel, whereby a portion of the service consumers get access to the new version, while the rest of the service consumers are still using the old version. A/B Releases are used to validate the new version in the productive environment, before fully releasing it to all consumers. In this scenario, the requests are load balanced by the Openshift Route to either the new or old service version.

PROJ_2 of Figure 9.4 illustrates a Blue/Green Release, which is used to run the old and the new version of a service in parallel, whereby the Openshift Route switches between the old and new service version. In this scenario, all service consumer access the same service version, which is currently accessible by the Openshift Route.

PROJ_3 of Figure 9.4 illustrates two service versions running completely separated in parallel, whereby both versions are accessible via their own Openshift Route. This scenario is not a release scenario, and should be used if multiple service versions have to be provided outside of the scope of a release.

Table 9.4 compares the Openshift and JBoss EAP contained mechanisms, which provide the listed release features. Multiple service versions on an Openshift Cluster are represented by separate Openshift Service-Objects, whereby with JBoss EAP, multiple service versions are either represented by separate JBoss EAP instances, or deployment contexts on a single JBoss EAP instance. With JBoss EAP, version switching and weighted routing can be performed by an external reverse proxy, which in Openshift can be performed by an Openshift Route.

Feature	**Openshift**	**JBoss EAP**
Multiple Service Versions	Openshift Service Openshift Route	Multiple EAP instances Multiple deployment contexts
Weighted Routing	Openshift Route	None (external)
Version Switch	Openshift Route	None (external)

Table 9.4: Release feature comparison

JBoss EAP is an application platform for Java applications, which provides an runtime environment for Java applications, but no networking features, as shown in Table 9.2 on page 61, and therefore no routing feature. JBoss EAP can be configured to act as a reverse proxy, but cannot act as the reverse proxy for applications hosted on the same JBoss EAP instance.

As Figure 9.4 on the preceding page illustrates, the implemented prototype can run multiple versions of the hosted services, and supports the implementation of release models such as A/B or Blue/Green Releases.

9.5 Managing Migration of Public API

This section will discuss the management of services public API, which is crucial, when it comes to distributed services. Changes made on the public API of a service can break the functionality of an application, the service is part of, or break the functionality of an external application, which depends on this service. The public API is the API the service exposes, which could only be consumed by internal consumers, but has to be managed the same way, as the API would be exposed to external consumers. There are several ways to migrate a public API, whereby some of them will be discussed in the further sections.

Services running on an Openshift Cluster are completely separated from each other and have their own life-cycle. Therefore, they have to provide a public API, which can be consumed by other services. The public API has to be designed properly in the first place, as well as the management of its migrations. At least, the last both versions will have to be supported, to give the developers of the consuming services enough time to apply to the migrations performed on the public API. Commonly, a service has two versions, on the one hand, the service release version, and on

the other hand, the service API version, which is allowed to remain the same over multiple service releases. The following sections will discuss the different ways to migrate the public API of a service.

URL Query-Parameter Versioning

Multiple versions of a public API can be managed via a URL Query-Parameter, whereby the URL Query-Parameter defines the version to use of a single API Operation. The URL `http://localhost/api/users?version=2` illustrates how to access a specific version of an API Operation via a URL Query-Parameter. The following Table 9.5 shows the Pros and Cons of API Versioning with URL Query-Parameters.

Pros	Cons
Single resource address	Data and control parameters are mixed
Supported by all browsers	Version as data parameter not possible
Easy to understand	No declarative mapping with JAX-RS
Easy to implement	Manual switch between API Operations

Table 9.5: Pros and Cons of URL Query-Parameter Versioning

HTTP Header Versioning

Multiple version of a public API can be managed via HTTP Headers in the following listed ways:

- New HTTP Header
  e.g. `Version: 2`
- Additional field in the Accept-Header
  e.g. `Accept: application/json; version=2`
- Enhanced Media-Type
  e.g. `Accept: application/vnd.app.model.v1+json;qs=0.9`

The approach of HTTP Header Versioning brings in more flexibility, but also makes the API Versioning harder to understand for consumers, especially when quality of service is used in the HTTP Accept-Headers of multiple API Operations. The following Table 9.6 shows the Pros and Cons of API Versioning with HTTP Headers.

Pros	Cons
Single resource address	Header handling needed
No mix of data and control parameters	Harder to implement/understand
Declarative mapping with JAX-RS	No enhanced Media-Type in HTML
Easy to implement	More difficult to test

Table 9.6: Pros and Cons of HTTP Header Versioning

Path Versioning

The easiest way to manage multiple versions of a public API is Path Versioning, whereby the whole API is versioned, instead of single API Operations. The URL `http://localhost/api/v2/users` illustrates how to access a specific version of an API Operation via a Path version. The Path Versioning is the most used approach to version a public API, because of its simplicity to realize. The following Table 9.7 shows the Pros and Cons of API Versioning with Path versions.

Pros	**Cons**
Easy to implement	Multiple resource addresses
Easy to understand	Declarative mapping with JAX-RS
No mix of data and control parameters	Version actually not part of resource
Easy to switch between versions	Latest version unknown

Table 9.7: Pros and Cons of Path Versioning

The prototype uses Swagger for documenting the services public API, as discussed in Section 8.7 on page 50, whereby Swagger is used to generate the Swagger Specification and clients. One advantage of a Swagger generated client is, that developers work with a generated class or interface, and therefore, developers work with a typed client, which will cause compile errors on breaking API changes. The API changes can be performed in a way, whereby developers will not have to change anything in their source code, but could also be performed in a way to cause compile errors, which force the developers to apply to the breaking API changes. If the client is implemented by hand and not generated by a tool, then developers will have to ensure that the implemented client is in sync with the currently used API version.

9.6 Managing Adapters and Message Translator as Services

Adapters and Transformers are part of the EI Patterns, whereby the Adapter is used couple an application to a message bus, and the Transformer is used to transform messages from the application supported format to the message bus supported format and visa versa. The Adapter and Message Translator can either be located at the external service, or can be located in the messaging system, the external service connects to. Commonly, a message bus system is implemented to use one form of communication, such as REST, and one form of data representation. If external services need to access the message bus, but don't support the message bus protocol and data format, then Adapters and Message Translators are needed, to connect the external service to the message bus.

Figure 9.5 on the next page illustrates how the prototype could be integrated with external applications via an Adapter and Message Translator. If the Adapter and Message Translator are located at the external service, then the Adapter and Message Translator have to be provided in the programming languages, the external

services are implemented with. If the Adapter and Message Translator are located in the Openshift Project, then they can be implemented in the programming language of choice.

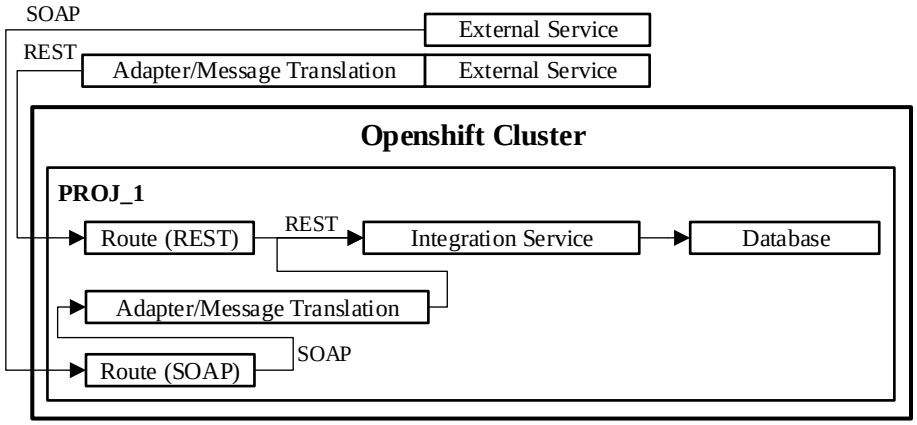

Figure 9.5: External Services connected via Adapter

The Adapter and Message Translator have to be implemented as a standalone application (microservice), which for instance can be realized in Java with the Camel framework, which provides an API for implementing EI Patterns, and is capable of running as a standalone application. The adapter and message translator application can be hosted as a separate Openshift Service, or can be added to the service Pod as a so called side-by container, which proxies the requests to the actual container.

9.7 Further Work

During writing this thesis and the implementation of the prototype, Red Hat has released its new version of the JBoss Fuse platform, which now is based on Openshift, and is called Red Hat Fuse 7[1]. Red Hat Fuse 7 enhances Openshift with features for implementing an ESB, whereby the main goal is to provide non-technical personal the possibility to create integrations. Red Hat Fuse 7 provides iPaaS features, by providing a low/no code UI, which allows to create integrations without the need to write a single line of code.

The next step is to implement an ESB with Red Hat Fuse 7, to see the differences between the manual approach, represented by the prototype, and the approach using iPaaS provided features. Red Hat Fuse 7 should make it easier for enterprises to create integrations to external services, as well as the management of these integrations.

[1] https://www.redhat.com/en/technologies/jboss-middleware/fuse

References

Literature

[Cor14] Intel Corporation. *Linux* Containers Streamline Virtualization and Complement Hypervisor-Based Virtual Machines*. Intel Corporation, 2014 (cit. on p. 15).

[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, 2, 26, 29).

[IA18] Claus Ibsen and Jonathan Anstey. *Camel in Action*. 2nd ed. Shelter Iland, NY: Manning Publications Co., 2018 (cit. on p. 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. 4–7).

[Liu+15a] Lawrence Liu et al. *Integration Platform as a Service: The next generation of ESB, Part 1*. IBM, 2015 (cit. on pp. 1, 21, 29).

[Liu+15b] Lawrence Liu et al. *Integration Platform as a Service: The next generation of ESB, Part 2*. IBM, 2015 (cit. on pp. 21, 29).

[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 pp. 29, 31).

[Sch14] Mathijs Jeroen Scheepers. *Virtualization and Containerization of Application Infrastructure: A Comparison*. University of Twente, 2014 (cit. on p. 9).

Online sources

[Ama18a] Amazon Web Services (AWS). *Amazon EKS*. 2018. URL: https://aws.amazon.com/eks/ (visited on 03/31/2018) (cit. on p. 16).

[Ama18b] Amazon Web Services (AWS). *AWS Elastic Beans Talk*. 2018. URL: https://aws.amazon.com/elasticbeanstalk/ (visited on 04/12/2018) (cit. on p. 21).

[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. 16).

[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. 19).

[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. 14).

[Doc18a] Docker Inc. *Docker Machine Overview*. 2018. URL: https://docs.docker.com/machine/overview/ (visited on 03/17/2018) (cit. on p. 13).

[Doc18b] Docker Inc. *Dockerfile Reference*. 2018. URL: https://docs.docker.com/engine/reference/builder/ (visited on 03/09/2018) (cit. on p. 10).

[Doc18c] Docker Inc. *Dockerfile Registry*. 2018. URL: https://hub.docker.com/ (visited on 03/09/2018) (cit. on p. 10).

[Doc18d] Docker Inc. *Overview of Docker Compose*. 2018. URL: https://docs.docker.com/compose/overview/ (visited on 02/23/2018) (cit. on p. 7).

[Doc18e] Docker Inc. *What is Docker*. 2018. URL: https://www.docker.com/what-docker (visited on 02/23/2018) (cit. on p. 9).

[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. *Eclipse MicroProfile*. 2018. URL: https://microprofile.io/ (visited on 06/05/2018) (cit. on p. 33).

[Ecl18c] Eclipse Foundation. *Fault Tolerance*. 2018. URL: https://github.com/eclipse/microprofile-fault-tolerance (visited on 06/05/2018) (cit. on p. 35).

[Ecl18d] Eclipse Foundation. *Metrics*. 2018. URL: https://github.com/eclipse/microprofile-metrics (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. 21).

[Goo18b] Google Cloud. *Kubernetes Engine*. 2018. URL: https://cloud.google.com/kubernetes-engine/ (visited on 03/31/2018) (cit. on p. 16).

[Heo15] Heo, Tejun. *CGROUPS*. 2015. URL: https://www.kernel.org/doc/Documentation/cgroup-v2.txt (visited on 03/30/2018) (cit. on p. 15).

[Hoh03] Hohpe, Gregor. *Get Started with Red Hat JBoss Fuse 7 Tech Preview 3 Today!* 2003. URL: https://www.enterpriseintegrationpatterns.com/ramblings/03_hubandspoke.html (visited on 04/12/2018) (cit. on p. 29).

[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. 18).

[Kub18b] Kubernetes. *Kubernetes Components*. 2018. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 04/01/2018) (cit. on pp. 19, 20).

[Kub18c] Kubernetes. *Pods*. 2018. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod/ (visited on 04/01/2018) (cit. on p. 18).

[Kub18d] Kubernetes. *Production-Grade Container Orchestration*. 2018. URL: https://kubernetes.io/ (visited on 03/30/2018) (cit. on p. 16).

[Kub18e] Kubernetes. *Secrets*. 2018. URL: https://kubernetes.io/docs/concepts/configuration/secret/ (visited on 04/01/2018) (cit. on p. 18).

[Kub18f] Kubernetes. *Secrets*. 2018. URL: https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/ (visited on 04/01/2018) (cit. on p. 18).

[Kub18g] Kubernetes. *Services*. 2018. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 04/01/2018) (cit. on p. 18).

[Lin18a] Linux Containers. *Linux Containers*. 2018. URL: https://linuxcontainers.org/ (visited on 03/29/2018) (cit. on p. 9).

[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. 15).

[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. 15).

[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. 21).

[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. 16).

[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. 22).

[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. 21).

[Ope18b] Openshift. *Build Strategies*. 2018. URL: https://docs.openshift.com/container-platform/3.9/dev_guide/builds/build_strategies.html (visited on 04/07/2018) (cit. on p. 24).

[Ope18c] Openshift. *Builds and Image Streams*. 2018. URL: https://docs.openshift.com/container-platform/3.9/architecture/core_concepts/builds_and_image_streams.html#image-streams (visited on 04/07/2018) (cit. on p. 25).

[Ope18d] Openshift. *How Deployments work ?* 2018. URL: https://docs.openshift.com/container-platform/3.9/dev_guide/deployments/how_deployments_work.html (visited on 04/07/2018) (cit. on p. 25).

[Ope18e] Openshift. *OpenShift Application Platform*. 2018. URL: https://github.com/openshift/origin (visited on 04/12/2018) (cit. on p. 22).

[Ope18f] Openshift. *Overview*. 2018. URL: https://docs.openshift.com/container-platform/3.9/architecture/core_concepts/ (visited on 04/07/2018) (cit. on p. 22).

[Ope18g] 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. 36).

[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. 7).

[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. 21).

[Red18b] Red Hat Inc. *Integrated Development Platform*. 2018. URL: http://fabric8.io/ (visited on 07/02/2018) (cit. on p. 40).

[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).

[Red18f] Red Hat Inc. *Switchyard*. 2018. URL: http://switchyard.jboss.org/ (visited on 06/30/2018) (cit. on p. 31).

[Sma18] SmartBear Software. *Swagger*. 2018. URL: https://swagger.io/ (visited on 06/05/2018) (cit. on p. 36).

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