Implementation of an Enterprise Service Bus with OpenShift and Camel

Ing. Thomas Herzog B.Sc

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Software Engineering

in Hagenberg

im Juni 2018

Declaration

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

Hagenberg, June 1, 2018

Ing. Thomas Herzog B.Sc

Contents

Declaration iii

Preface v

Abstract vi

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2

2 Infrastructure as Code (IaC) 3
- 2.1 The Need for IaC . . . 3
- 2.2 Principles of IaC . . . 5
  - 2.2.1 Infrastructures are Reproducible . . . 5
  - 2.2.2 Infrastructures are Disposable . . . 5
  - 2.2.3 Infrastructures are Consistent . . . 6
  - 2.2.4 Actions are Repeatable . . . 6

3 Containerization with Docker 7
- 3.1 The need for Containerization . . . 7
- 3.2 Docker . . . 8
  - 3.2.1 Docker Engine . . . 8
  - 3.2.2 Docker Architecture . . . 9
- 3.3 Virtualization vs. Containerization . . . 9

References 11
- Literature . . . 11
- Online sources . . . 11

Preface

Abstract

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

Chapter 1

Introduction

1.1 Motivation

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

The major challenge of an IT department is the integration of independent applications into the enterprise application environment. The concept of Enterprise Application Integration (EAI) provides patterns [Hohpe and Woolf 2008], 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.

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 [Red Hat Inc. 2018a], which provides tooling to integrate applications into an ESB. JBoss Fuse is based on the JBoss Enterprise Application Platform (JBoss EAP), where the applications are integrated in a existing runtime environment.

With the upcoming of cloud solutions such as Platform as a Service (PaaS) [Devi and Ganesan 2015, p. 2-3] it is now possible to move the platform from a dedicated environment to a cloud environment, where each integration service has its own runtime environment rather than joining an existing runtime environment. The concept of Integration Platform as a Service (IPaaS) [Liu et al. 2015, p. 3] relies on top of PaaS and enhances a common PaaS solution with the Integration features needed by EAI.

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

1.2 Objectives

This thesis aims to implement an ESB on Openshift PaaS [Red Hat Inc. 2018b]. Commonly an ESB is implemented with the help of middleware such as JBoss Fuse, which is based on the JBoss EAP. The concepts of PaaS and IPaaS are in general new to the industry, which commonly hosts their integration services in their own data centers, due to the lack of trust for cloud solutions and knowledge about the new approaches such as microservice architecture.

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

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

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

Chapter 2

Infrastructure as Code (IaC)

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 [Kief Morris 2016, p. 5]. IaC descriptions are called templates, cookbooks, recipes or playbooks, depending on the tool. In the further course, the IaC definitions will be called templates. The DSL allows to define resources of a system such as network, storage and routing descriptively in a template. The DSL abstracts the developer from system specific settings and provides a way to define the system with as little configuration as possible. The term system is used as a general description. In the context of IaC, a system can be anything which can be described via a DSL.

2.1 The Need for IaC

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 [Kief Morris 2016, p. 4]. The IT systems are decoupled from the physical hardware and operating system, due to the fact, that cloud providers cannot allow their customer to tamper with the underlying system and hardware. In general, the hardware resources provided by a cloud provider are shared by multiple customers.

With IaC it is possible to work with so called dynamic infrastructure platforms, which provide computing resources, where the developers are completely abstracted from the underlying system. Dynamic infrastructure platforms have the characteristic to be programmable, are available on-demand and provide self service mechanisms, therefore we need IaC to work with such infrastructures [Kief Morris 2016, p. 21-22]. Systems deployed on a dynamic infrastructure platform are flexible, consistent, automated and reproducible.

Enterprises which stuck to legacy systems face the problem that technology nimble competitors can work with their infrastructures more efficiently, and therefore can

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

Figure 2.1: Automation Fear Spiral

Because of no trust for the automation, changes are applied manually to the systems and outside the defined automation process. If the system is reproduced, definitions may be missing in the templates, which leads to an inconsistent system. Therefore, enterprises have to break this spiral to fully profit from IaC [Kief Morris 2016, p. 9].

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

2.2 Principles of IaC

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

2.2.1 Infrastructures are Reproducible

With IaC, systems are easy reproducible. It is possible to reproduce the whole infrastructure or parts of it effortlessly. Effortless means, that no tweaks have to be made to the templates or during the reproduction process and there is no need for a long term decision process about what has to be reproduced and how to reproduce it. To be able to reproduce system effortlessly is powerful, because it can be done automatically, consistently and with less risk of failures [Kief Morris 2016, p. 10]. The reproducibility of a system is based on reusable templates which provide the possibility to define parameters, which are set for the different environments as shown in figure 2.2.

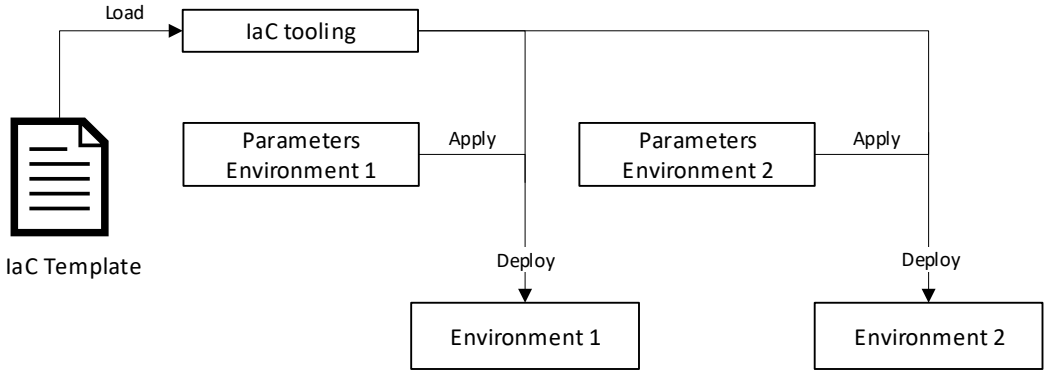

Figure 2.2: Schema of a parametrized infrastructure deployment

2.2.2 Infrastructures are Disposable

Another benefit of IaC is that systems are disposable. Disposable means, that systems can be easily destroyed and recreated. Changes made to the templates of a system does not have to be applied on an existing system, but can be applied by destroying and recreating the system. An requirement for a disposable system is, that it is understood that systems will always change. Other systems relying on a disposable system need to address that the system could change at any time. Systems must not fail because a disposable system disappears and reappears again because of an redeployment [Kief Morris 2016, p. 11].

2.2.3 Infrastructures are Consistent

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

In listing 1 an example for an IaC template is shown, which defines a Docker Compose [Docker Inc. 2018b] service infrastructure for hosting a Wildfly [Red Hat Inc. 2017] server instance. This system can consistently be reproduced on any environment supporting Docker, Docker Compose and providing values for the defined parameters.

Listing 1: Example for an IaC template for Docker Compose

```
version: "2.1"
services:
   wildfly:
        container_name: wildfly
        image: wildfly:latest
        depends_on:
            - postgres
        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}
```

2.2.4 Actions are Repeatable

Building reproducible systems, means that any action applied to the system should be repeatable. Without repeatability, the automation cannot be trusted and systems wouldn't be reproducible. An instance of a system in another environment should be equal to any other system instance, except for the configurations defined by parameters. If this is not the case, then a system is not reproducible, because it will have become inconsistent [Kief Morris 2016, p. 12 -13].

IaC is a concept which makes it very easy to deal with systems in the cloud age. Enterprises can make use of IaC to move their legacy systems to the cloud, where they can profit from the principles of IaC. Nevertheless, before an enterprise can profit from IaC, it has to apply clear structures to their development process, as well as sticking to the principles of consistency and repeatability. For experienced administrators, who are used to maintain systems manually, it could sometimes be hard to understand why they are not supposed to perform any actions on the system manually anymore, nevertheless that a manual change could be performed faster. Being capable to reproduce a system at any time with no effort, or applying changes on an existing system in a predefined and consistent manner, makes enterprises very flexible and fast. Enterprises will not have to fear future changes in requirements and technologies of their systems anymore.

Chapter 3

Containerization with Docker

Docker [Docker Inc. 2018c] 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 OS virtualization. The differences of LXC an hypervisor based virtualization a re covered in section 3.3. Docker has become very popular over the past years, due to the fact, that it made it possible to easily work with LXC. Docker relies strongly on the principles of IaC which has been discussed in chapter 2. When using Docker, Linux containers are often referred as Docker containers.

Containerization is a key factor in cloud based development, because applications normally are 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 are discussed in section 3.1.

3.1 The need for Containerization

Containerization is a key factor for cloud platforms such as PaaS, where each application runs in its own isolated environment, called a container. A container is an instance of an image, which represents the initial state of an application. A virtual machine represents a full blown OS, where the OS provides a kernel, which is emulated on the host OS by the hypervisor. A hypervisor is a software which can create, run and manage virtual machines. A container uses the kernel provided by the host OS and therefore does not perform any emulation. A container does not represent a full blown OS, but still provides features such a networking, storage and the OS file system.

Containers are faster to create, deploy and easier to manage than virtual machines. Nevertheless, cloud platforms use virtualization for managing their infrastructure, where the containers run on the provisioned virtual machines. The usage of containers compared to the usage of virtual servers can reduce costs for hosting applications. Enterprises can profit from hosting their applications of containers in several ways. Applications hosted in containers need lees resources than applications hosted in virtual servers, because there is no virtualized OS. The creation, deployment and startup of containers are faster, because only the isolated process needs to be started and not a full blown OS. Docker is well supported by Integrated Development Environments (IDEs), which provide support for creating Docker image definitions (Dockerfiles) and provisioning of

Docker Containers on a local or remote environment.

When enterprises have applied IaC to their infrastructure life cycle, then the next logical step is to integrate their applications into IaC as well. Applications hosted in containers profit from the IaC principles immutability, reproducibility, repeatability and consistency. Therefore, Docker strongly relies on IaC and provides tooling for automating creation and deployment of docker containers, which is used by PaaS platforms such as Openshift. With Docker, developers define the hosting environment for their applications now and not system administrators anymore. Nevertheless, developers can profit from the deep Linux knowledge of system administrators, to define the Docker images efficiently, to keep them small and secure.

3.2 Docker

This section covers Docker, which is the most popular tool to work with LXC. Docker is open source but also provides an enterprise support if needed. The core part of the Docker technology is the Docker Engine which is covered 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 [Docker Inc. 2018a], which is a repository for Docker images. The most popular Docker registry is Docker Hub [Docker Inc. 2018a], which is a free service, where anyone can provides Docker images.

3.2.1 Docker Engine

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

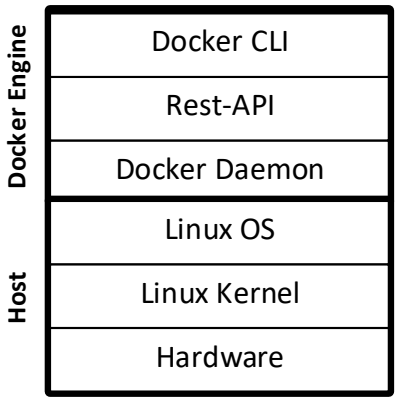

Figure 3.1: Docker Engine Architecture

Docker Damon and REST-API

The docker daemon represents the background process on a Linux OS, which creates, runs and manages the Docker Containers on the Docker host. A connection to the Docker daemon can be established via a Unix socket or a network interface, depending on the configuration of the Docker daemon. If the Docker daemon is exposed via a network interface, then it is recommended to secure the connection with client certificate authentication.

Docker Command Line interface (CLI)

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

Docker Images

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

Docker Containers

A Docker container is an instance of a Docker image, where a new layer is appended, which contains all changes made on the file system by the running process within the Docker container. When the container is deleted, then the appended layer gets deleted as well and all made changes on the file system are lost. A Docker container keeps running as long as the contained foreground process is running. Without a foreground process the Docker container stops immediately after it was started. The process running in the container is isolated from other processes, as well is the file system, the process has access to.

3.2.2 Docker Architecture

The Docker architecture is a client server architecture, therefore the communication to the Docker daemon is done via the REST-API.

3.3 Virtualization vs. Containerization

Before LXC, the industry made heavy use of operating system (OS) virtualization to isolate their environments and applications. A virtual machine is managed by a hy-

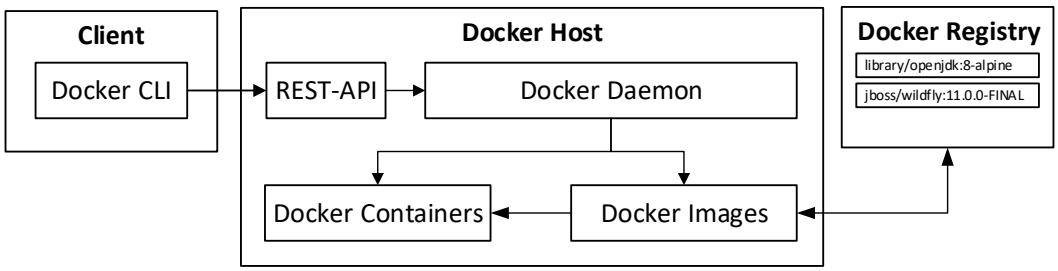

Figure 3.2: Docker Architecture

pervisor, which is software, which can create, run and manage virtual machines. The virtual machine provides resources such as network and storage for the application, which is managed by the virtualized OS. Nevertheless, an virtual machine represents a full blown OS, which itself has a resource need which adds to the resource needs of the hosted application.

LXC on the other hand is a kernel technology, which provides resources such as network and storage to the application as well, but without the need of virtualized OS.

References

Literature

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

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

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

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

Online sources

Docker Inc. (2018a). *Docker Registry*. URL: https://docs.docker.com/registry/ (visited on 03/09/2018) (cit. on p. 8).

— (2018b). *Overview of Docker Compose*. URL: https://docs.docker.com/compose/overview/ (visited on 02/23/2018) (cit. on p. 6).

— (2018c). *What is Docker*. URL: https://www.docker.com/what-docker (visited on 02/23/2018) (cit. on p. 7).

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

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

— (2018b). *Red Hat Openshift Container Platform*. URL: https://www.openshift.com/container-platform/features.html (visited on 02/22/2018) (cit. on p. 2).

List of Figures

2.1 Automation Fear Spiral 4
2.2 Schema of a parametrized infrastructure deployment 5

3.1 Docker Engine Architecture 8
3.2 Docker Architecture 10

List of Listings

1 Example for an IaC template for Docker Compose 6