Implementation of an Enterprise Service Bus with OpenShift and Camel

Ing. Thomas Herzog B.Sc

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Software Engineering

in Hagenberg

im Juni 2018

Declaration

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

Hagenberg, June 1, 2018

Ing. Thomas Herzog B.Sc

Contents

Preface

Abstract

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

Chapter 1

Introduction

1.1 Motivation

Large enterprises work with several independent applications, where each application covers an aspect of a business of an 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 part of a business workflow. The applications have to interexchange data, which is commonly represented in different data formats and versions. This leads to an 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 connects multiple 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 helps 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 the cloud provider will take over responsibility for maintenance and security of the platform. The concept of Integration Platform as a Service (IPaaS) [Liu et al. 2015, p. 3] is on top of PaaS and enhances a common PaaS solution with the Integration features needed by EAI.

Thus, enterprises can now reduce the effort in implementing an ESB, integrating applications into the ESB and reducing the costs of 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 in combination with an ESB are in general new to the industry. In general the industry hosts their ESBs in their own data centers, due to the lack of trust for cloud solutions.

A main focus of this thesis is how applications internal and external can be integrated and managed in the PaaS solution Openshift. The different aspects of applications integrated into an ESB such as security, staging and versioning of will also be discussed in this thesis. The security is commonly managed by a middleware, which is in general hosted on a dedicated data center owned by the enterprise. With the usage of PaaS such as Openshift the responsibility of the security mostly shifts to the cloud provider.

// TODO: Futher writing

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 playbook depending on the tool. In the further course, the IaC definitions will be called templates. Templates allow to define resources of a system such as network, storage and routing. 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. A system can be a server or a service structure of an PaaS application.

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 to 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 to 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 in the automation, changes are applied to the systems manually and outside the defined automation process. Thus, the systems can become inconsistent, because changes maybe haven't been added to the system templates. 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 in 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 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.

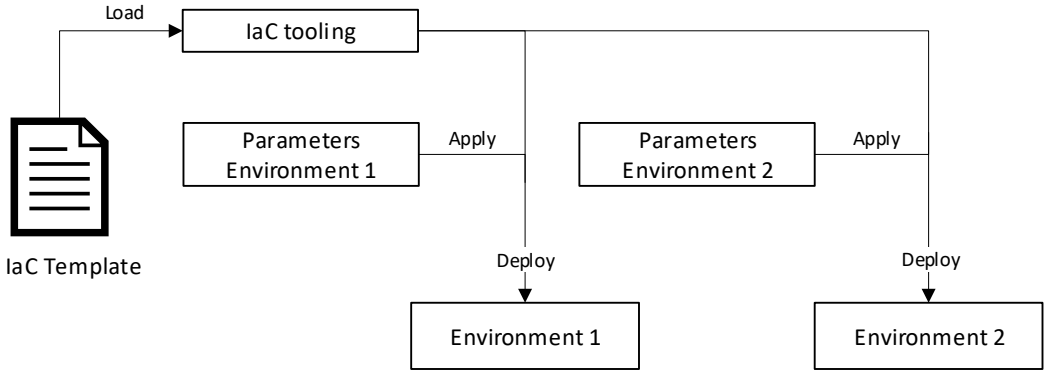

Figure 2.2: Schema of a parametrized infrastructure deployment

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 from 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. 2018] 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 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

3.1 The need for Containerization

3.2 Containerization Concepts

3.3 Virtualization vs. Containerization

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

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

List of Listings