

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Miika Piironen

Containerization and Cloud Migration of Legacy Web Services

Master's Thesis
Espoo, May 25, 2020

Supervisors: Senior University Lecturer Vesa Hirvisalo
Advisor: M.Sc. (Tech.) Jaakko Kotimäki
D.Sc. (Tech.) Mikko Hakala

Aalto University

School of Science

 Master's Programme in Computer, Communication and
 Information Sciences

 ABSTRACT OF
 MASTER'S THESIS

Author:	Miika Piironen		
Title:	Containerization and Cloud Migration of Legacy Web Services		
Date:	May 25, 2020	Pages:	50
Major:	Computer Science	Code:	SCI3042
Supervisors:	Senior University Lecturer Vesa Hirvisalo		
Advisor:	M.Sc. (Tech.) Jaakko Kotimäki D.Sc. (Tech.) Mikko Hakala		
<p>A research group has multiple web services running on an outdated server hardware. Many of the services are old and not actively developed anymore and also often depend on outdated software, which is problematic from the security point of view. It is time to decommission the old hardware and therefore the services needs to be migrated onto a more modern platform. While migrating away from the old servers, we want to make the services easier for the research group to maintain and improve security where possible.</p> <p>Containerization technologies are an increasingly popular way to build, package and deploy software. Containers provide a convenient way to package software along with it dependencies to be easily run across different computers and operating systems. While being more lightweight than virtual machines, containers provide a layer of isolation between services running on a same host. Containerized services can be hosted on a cloud container platforms such as Kubernetes or OpenShift.</p> <p>In this thesis work, multiple existing web services built on top of varying techonologies are containerized. The containerized services are then deployed onto an OpenShift cloud container platform. We see how containerization can lead to better maintainability and security of olded services. Containerization provides a layer of isolation between the services improving security and makes it easier to deploy them on different plaftorms if needed. The OpenShift platform provides container orchestration and tools for automating builds and deployment, which we utilize to make sure that the services and their dependencies are always kept upt-do-date.</p>			
Keywords:	cloud computing, containerization, migration, openshift		
Language:	English		

Aalto-yliopisto

Perustieteiden korkeakoulu

Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN

TIIVISTELMÄ

Tekijä:	Miika Piironen		
Työn nimi:	Legacy -palveluiden kontainerisointi ja pilvimigraatio		
Päiväys:	25. toukokuuta 2020	Sivumäärä:	50
Pääaine:	Tietotekniikka	Koodi:	SCI3042
Valvojat:	Vanhempi yliopistonlehtori Vesa Hirvisalo		
Ohjaaja:	DI Jaakko Kotimäki TkT Mikko Hakala		
<p>Tutkimusryhmällä on useita verkkopalveluita, joita ajetaan vanhentuneella, paikallisella palvelinalustalla. Monet näistä palveluista eivät ole enää aktiivisessa kehityksessä. Useat palveluista ovat riippuvaisia vanhoista ohjelmistoversioista, joka on ongelmallista palveluiden turvallisuuden kannalta. Vanhat palvelimet on tarkoitus poistaa käytöstä, ja palvelut tulee siirtää uudelle alustalle. Siirron yhteydessä haluamme tehdä palveluista tutkimusryhmälle helpompia ylläpitää sekä parantaa palveluiden turvallisuutta, mikäli mahdollista.</p> <p>Kontainerisointi on suosittu tapa rakentaa, paketoita ja ajaa ohjelmistoja. Kontainerit mahdollistavat ohjelmiston ja sen riippuvuuksien sisällyttämisen samaan pakettiin, tehden ohjelmiston ajamisesta eri alustoilla ja käyttöjärjestelmillä helpoa. Kontainerit ovat kevyempiä kuin perinteiset virtuaalikoneet, mutta eristävät kuitenkin samassa ympäristössä ajettavat ohjelmistot toisistaan. Kontainerisoituja palveluita voidaan ajaa pilvikontaineriympäristöissä, kuten Kubernetes ja OpenShift.</p> <p>Tässä diplomityössä kontainerisoinimme useita eri teknologioihin perustuvia verkkopalveluita. Kontainerisoidut palvelut viedään OpenShift -pilvikontainerialustalle. Näemme, kuinka kontainerisointi voi parantaa verkkopalveluiden ylläpidettävyyttä ja turvallisuutta. OpenShift -alusta huolehtii kontainereiden orkestroinnista sekä tarjoaa meille työkalut kontainereiden luonnin ja ajamisen automatisointiin. Hyödynnämme tätä palveluiden ja niiden ohjelmistoriippuvuuksien automaattiseen päivitykseen.</p>			
Asiasanat:	pilvilaskenta, kontainerisointi, migraatio, openshift		
Kieli:	Englanti		

Espoo, May 25, 2020

Miika Piironen

Abbreviations and Acronyms

CI/CD	Continuous Integration and Continuous Deployment
SSH	Secure Shell
CLI	Command Line Interface
DNS	Domain Name System
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
CGI	Common Gateway Interface
URL	Universal Resource Locator
VCL	Varnish Configuration Language
YAML	YAML Ain't Markup Language
SSL	Secure Sockets Layer
PV	Persistent Volume
PVC	Persistent Volume Claim
API	Application Programming Interface
VM	Virtual Machine
SOA	Service Oriented Architecture
HTTP	HyperText Transfer Protocol
IO	Input/Output
PID	Process Identifier
ID	Identifier
UID	User Identifier

Contents

Abbreviations and Acronyms	5
1 Introduction	8
1.1 Scope	8
1.2 Problems	9
1.3 Results	9
1.4 Structure of the thesis	10
2 Background	11
2.1 Software process	11
2.2 Cloud computing	12
2.3 Cloud migration	13
2.4 Service based architectures	13
3 Key technologies	15
3.1 Containerization	15
3.2 Docker	16
3.3 OpenShift	17
3.4 OpenShift objects	17
3.4.1 Deployments	18
3.4.2 Services	18
3.4.3 Images and ImageStreamTags	18
3.4.4 Builds	19
3.4.5 Volumes	19
3.4.6 ConfigMaps	19
3.4.7 Routes	19
3.5 Docker Hub	20
3.6 GitHub	20
3.7 GitLab	20

4	Implementation	21
4.1	Architecture	21
4.1.1	Frontend	21
4.1.2	Web server	22
4.1.3	App server	22
4.1.4	Store server	22
4.1.5	Changes	22
4.2	Containerization	23
4.2.1	Varnish Cache	24
4.2.2	Apache websites	25
4.2.3	Wordpress sites	26
4.2.4	Tomcat applications	26
4.2.4.1	Java version	27
4.2.5	Fuseki databases	27
4.3	Git repositories	28
4.4	Automating Builds and Deployment	29
4.5	Backups	30
4.6	Routing	32
4.7	Migrating Data	33
5	Discussion	39
5.1	Splitting cache and database	39
5.2	Benefits from PaaS	40
5.3	Standardized respositories	40
5.4	Service isolation	40
5.5	Automated deployment	41
5.6	Scalability	41
5.7	Amount of work	42
5.8	Code duplication	42
5.9	Multiple processess per container	43
5.10	Making containers more generic	43
5.11	Storage problems	44
5.12	Monitoring	44
5.13	Testing	44
5.14	Cost Optimization	45
6	Conclusions	46

Chapter 1

Introduction

Containerization technologies makes it easier to deploy software across across varying platforms and operating systems. The software is packaged with it's dependencies and libraries and is guaranteed to run in the same way on a local development machine or a production server. Containerization is an increasingly popular way to build, package and deploy software

Cloud Computing is a computing model that frees software developers and IT operations from managing computing resources. Instead of using dedicated server hardware, software can be deployed on a cloud platform with a pool of shared computing resources. Many public cloud providers also offer platforms allowing deployment and nearly infinite scalability of containerized software. Therefore it is no wonder that big portion of the newly developed software is utilizing containers in some form.

In this thesis, we are applying containerization technologies and cloud computing resources in migration of existing web services. We are moving the software services belonging to a reserch group from on-premises servers to an OpenShift cloud container platform. This involves reengineering of the software services for containerization and deploying the services on the cloud platform.

Next in this chapter, we detail the scope, problem and results of this work.

1.1 Scope

The services migrated in this work belong to the Semantic Computing Research Group (SeCo) of Department of Computer Science, Aalto University. The main goal of the work is to migrate these services, residing on physical on-premises servers, onto a cloud container platform.

Firstly, the work introduces the relevant research areas including Software Engineering, Cloud Computing, Service Architectures and Containerization. Secondly, this work explains the key technologies used and the implementation of the migration. Thirdly, the work discusses the outcomes and what can be learned from this migration project.

1.2 Problems

The on-premises server of the SeCo group have come to their end-of-life. Older the hardware gets, the more likely it is to break. Therefore we need to find a replacement for the hardware.

The operating system and software versions are outdated, which is a problem from the security point of view. Old software is likely to contain lots of security vulnerabilities. The outdated software should be updated or replaced. Possible security problems should be mitigated otherwise where updating or replacing software is not possible.

Deployment of the software services is on most cases not documented properly. Configuration code quality has degraded over time due to lack of refactoring. There is one configuration file for a cache handling traffic for all of the services, that is especially difficult to read and understand. This makes the configuration error-prone. All of these problems make the maintenance of the services difficult. In order to improve the maintainability of the services, we want to improve the documentation and refactor some of the configuration items.

Problem related to legacy systems and software modernization are a common topic in the software industry. As an example, Khadka et al.[11] interviewed 26 industrial practitioners on their views on legacy systems and software modernization.

1.3 Results

This work is based on a migration project conducted by the author. In the project, various web services are migrated from an on-premises servers to a containerized cloud environment. In total, there are around 30 services including simple websites, deployments of off-the-shelf applications, databases, Tomcat java applications and some custom pieces of software.

We demonstrate how the different application types can be containerized. We discuss the problems we encountered and how they were handled.

Based on the experiences from the migration project, we discuss: 1) How the containerization can improve security, 2) How reengineering for containers and cloud can improve maintainability, 3) Amount of work involved, 4) Mistakes and problems we encountered during the migration project and 5) Drawbacks of cloud and microservice architectures

1.4 Structure of the thesis

In this chapter, we introduced the scope, problem and results as well as the structure of the thesis. Chapter 2 introduces Software Process, Cloud Computing, Cloud Migration and Service Architectures and the relevant works. Chapter 3 introduces the key technologies used in the migration project: Containerization, Docker, OpenShift and Git. In Chapter 4, we go through the implementation of the migration. In Chapter 5 we discuss the results and experiences from the migration. Chapter 6 concludes the thesis.

Chapter 2

Background

In this chapter we introduce the relevant research and literature. The relevant research topics are introduced in the following order: software process, cloud computing, cloud migration and service architectures.

2.1 Software process

In his book, Sommerville[25] describes the idea of a *software process* - a set of activities for software production. Later part of the process is called software evolution or software maintenance. The software is in production and is mainly changed only to fix bug or vulnerabilities, to adapt to new environments or in order to add functionality to support new requirements.

Sommerville[25] also talks about dealing with legacy systems. In his words, "Legacy systems are older systems that rely on languages or technology that are no longer used for new systems development". Such systems become more difficult and expensive to maintain over time. Sommerville talks about different options for dealing with legacy systems: scrapping the system, continuing with regular maintenance, reengineering the system or replacing parts of, or the whole system. The choice mainly depends on the importance of the system and the related costs.

There are lots of studies done regarding to legacy systems and software modernizations. In their book, Seacord et al.[22] describes the software modernization process in a business context. In 2013, Khadka et al.[11] did an empirical study, interviewing 26 industry practitioners about their views on legacy systems and software modernization.

While in our case, there is not a well defined process for the development of the software systems we are dealing with in this thesis, these points are still highly relevant to the project. The motivation for the migration project

comes from the hardware that has come to the end of its lifecycle. However the software hosted using the hardware still needs to be maintained. Therefore there is a need to migrate and possibly adapt the software for a new platform.

There is also many pieces of software that rely on older technologies that might not be supported anymore. Therefore there is a need to evaluate what to do with each of these software. For most of the services, some degree of *reengineering* is needed in order to adapt them for the new platform. There are also some services that can be scrapped. There are also some parts of the software systems that we can easily replace with more modern alternatives.

2.2 Cloud computing

Cloud Computing is a wide concept. What is called a cloud, can vary from small computer clusters used and hosted by a small company, to world wide services consisting of multiple datacenters and hundreds of thousands of computers. The cloud services provided range from virtualized hardware to application platforms to software running on a web browser.

In 2010, Armbrust et al.[3] wrote a popular article about cloud computing. In the article they go through the main advantages of the cloud computing model. They also list the various obstacles and risks associated with cloud computing that might keep businesses from adopting cloud computing regardless of the advantages over investing to traditional, on-premises hardware.

One of the most widely referred definitions for Cloud Computing comes from NIST Definition of Cloud Computing by Mell and Grance[14]. In their definition of the cloud computing model, they identify the essential characteristics of a cloud service, different cloud service models as well as different cloud deployment models.

Rittinghouse and Ransome[21] have written an introductory book about cloud computing in 2016.

In its essence, cloud computing is a model, where a shared pool of computing resources are accessed via the network. The resources can be quickly provisioned and released on-demand. In practice this means that the user of a cloud service can, for example, create virtual machines or deploy their application code using a web interface, an API or other tools from the cloud provider.

The three main cloud service models are *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)* and *Software as a Service (SaaS)*. In IaaS, users can deploy virtualized hardware, most commonly virtual machines (VMs)

on-demand. In PaaS, the users are offered tools that they can use to deploy their applications on the cloud provider's infrastructure. In SaaS, the cloud provider offers an access to a software hosted on their infrastructure via the network.

2.3 Cloud migration

Zhao et al.[35] surveyed many papers on on-premises to cloud migration and provide an overview of different strategies. They categorize the migrations to three categories: migration to IaaS, migration to PaaS and migration to SaaS.

In migration to IaaS, the software is rehosted on cloud provider's infrastructure. This is usually the least involved strategy. Reengineering or changes to the application are usually not needed. Usually a virtual machine containing the application is built and hosted on the cloud provider's infrastructure.

Migration to PaaS is a more involved strategy. Instead of infrastructure, the cloud provider provides a platform to run the application on, such as a database and an environment to run code in. This usually requires adapting the software for the platform. The advantage over IaaS is in the resource management. We do not need to worry about allocating resources or provisioning/configuring virtual machines.

In migration to SaaS, the application or parts of it is replaced by a cloud alternative. This often means only copying/transforming data from the local system to the cloud alternative.

Our migration strategy is essentially *migration to PaaS*. We will be adapting the service software so that it can be run on the OpenShift platform. This includes creating the container images and configuration that enable deployment of the application on the OpenShift platform.

2.4 Service based architectures

Service Oriented Architecture (SOA) and *microservice architectures* are service based architectures, where applications are divided into smaller service components that communicate with each other over some network protocol. As described by Richards[20], the two architecture types are very different. SOA puts emphasis on sharing and reusing functionality whereas microservices aim to create self-contained components with minimal dependencies. However both architecture types have the advantage of distributed architec-

ture. The service components can be distributed over multiple servers and scaled individually as needed.

Service based architecture and containerization go hand-in-hand. Service based architecture can be seen as a requirement when developing for a cloud container platform such as OpenShift - or the platforms can be seen as enabling technologies for the service architectures. In the cloud, containers are essentially self-contained service components that communicate with each other and to the internet over the network.

All the applications migrated in this project already more or less conform to the service based architecture. The components such as databases, web applications, cache and other programs are more or less self contained and mainly communicate over the HTTP protocol. However we still need to identify the different components and make decisions on what to include in a single container, or essentially where the service boundaries are.

Chapter 3

Key technologies

In this chapter we introduce the key technologies used in the migration. Section 4.2 introduces containerization. Section 3.2 introduces one the most popular widely used container runtimes: Docker. Sections 3.3 and introduces the OpenShift cloud container platform and OpenShift objects that are used in configuring the platform. Section 3.5 introduces DockerHub, a public repository used for storing container images. Sections 3.6 and 3.7 introduce the code repositories GitLab and GitHub that we use for storing source code and deployment configuration for the services.

3.1 Containerization

Containerization is usually defined as a way to package software and its dependencies so that it can be run in varying computing environments. Containerized software is usually described as a more secure compared to regular software as the containerization provides a layer of isolation between different software running on the system. Containerization is also described as a lightweight alternative to hardware level virtualization or virtual machines. [6]

Containerized software is usually delivered as a container image. The image contains the filesystem with the software as well as any binaries, libraries and files the software depends on. The image also contains instructions for running the contains such as volume locations and network ports to expose. For running a containerized image, a container runtime is needed. The most widely known way to run containerized software is using Docker[5].

A *containerized process* usually means that the process 1) cannot see processes running outside of the container on the same host system, 2) cannot access files or filesystem of the host system, 3) cannot access devices

or network on the host system, 4) has limits on system resources, such as processor time and memory. In order to set up the containerized process like this, the container runtime utilizes 2 sets of linux kernel features: *control groups (cgroups)* and *namespaces*.

cgroups allows setting limits to a group of processes, including memory usage, IO bandwidth and CPU usage. *namespaces* are used to control access to different system resources. There exists namespaces for mounts, pids, uids, network, cgroups, etc (hostname), and ipc. Containerization includes creating all of these namespaces for a process or set of processes, but the most relevant are the first 4 types of namespaces. [5][29][28]

A *mount namespace* has its own set of mount points. This means that the processes cannot see filesystems or mounts from the host system or other namespaces. [5][29]

A *PID namespace* has its own set of process IDs. This means that processes in a PID namespace can only see processes within the same namespace. PID namespaces are nested, and processes have different ids for each namespace they belong to. [5][29]

An *UID namespace* has its own set of user IDs. UID namespaces are also nested and users get different ids in different namespaces. This means, for example, that a root user (pid 0) in a container shows up as a non-root user on the host system. [5][29]

A *network namespace* has its own network resources, such as network interfaces, routing table, firewall etc. [5][29]

Running a containerized process is a complicated process involving at least 1) creating the namespaces for the process. 2) Mounting filesystem in the mount namespace of the process. 3) Setting up virtual network interfaces in the process's network namespace for network access. The container runtime is an abstraction layer that makes running a containerized process using a single command. The specifics are governed by the defaults of the container runtime, instructions included in the container image and options given by the user.

3.2 Docker

The most widely known container runtime is *Docker*[1]. At its simplest, running a containerized software with Docker only requires the user to only specify the container image.

Additional configuration, such as user to run the containerized process as, networking, mounts or command to run can be defined when starting the container. It is usual to define additional mounts for configuration files

or persistent data or network ports to expose on the host system's network interfaces. [5]

For example, the following Docker command would run a containerized NGINX web server while mounting the content to serve from a location on the host system. The command also explicitly exposes the container's port 80 on the host system's network interface on port 8080.

```
docker run nginx -v /srv/www:/usr/share/nginx/html -p 8080:80
```

3.3 OpenShift

In this section we briefly introduce the relevant OpenShift features

OpenShift[19] is a Kubernetes based cloud container platform by Red Hat. It provides container orchestration similarly to Kubernetes, as well as lots of additional features providing tools for building, deploying, managing and monitoring. OpenShift abstracts the underlying hardware. In OpenShift we define objects, such as DeploymentConfig, ImageStreams, BuildConfigs, PhysicalVolumeClaim (PVC), Services and Routes. OpenShift builds, deploys, allocates persistent storage and defines networking based on these objects. For example, using DeploymentConfig we can define the container images and other options for deploying one or multiple containers. OpenShift takes care of deploying the containers and scheduling them on the underlying physical nodes. [18]

3.4 OpenShift objects

One of the most important concepts in Kubernetes are the *Kubernetes Objects*. Objects represent the *state* of the Kubernetes cluster including, for example, containers running in the cluster, services available in the cluster, storage and different policies. The objects contains fields called spec and state. spec is defined by the user and describes a desired state. state contains the actual state of the object. It is up to Kubernetes to bring the actual state to match the given spec. [18][13]

For example, a *Deployment object* is can be used to define a pod with containers and specific amount fo replicas to be run. A Service object can be used to define a network service backed by some given set of pods running in the cluster.

OpenShift is based on Kubernetes and works on the same principle, but adds more object types. For example a *Route object* can be used to expose Services in a specific hostname, port and url path. Instead of Deployment, we

can create `DeploymentConfig` objects that manage the Deployments and lets us, for example, define triggers for creating new deployments automatically and life cycle hooks.

3.4.1 Deployments

In OpenShift, containers are deployed by defining a *DeploymentConfig* object. A `DeploymentConfig` usually has at least the following information:

1. *container image(s)* to be deployed
2. *ports* to be exposed
3. *volumes* to be mounted inside the container(s)
4. *environment variables* to be passed to the container(s)
5. Amount of *replicas* to be deployed

After creating the `DeploymentConfig`, OpenShift runs the containers specified in the `DeploymentConfig`. All the objects, including `DeploymentConfigs`, represent a desired state. OpenShift always works to bring the cluster to the desired state. This means that, for example, a pod is automatically restarted if it fails.

3.4.2 Services

In OpenShift, pods are usually not targeted directly. Instead, *Service* objects are used to represent the pods. A `Service` object usually define at least 1) a name, 2) source and target ports and 3) backing pods. Traffic targeted to the `Service` will load balanced and sent to the backing pods. `Services` can also be used to abstract external services outside the cluster. [18]

3.4.3 Images and ImageStreamTags

Container images in OpenShift are represented by *ImageStream* objects. Similarly to docker registries, an `ImageStream` may contain multiple tags with different versions of the container images. `ImageStreams` are an abstraction over the actual image registries. `ImageStreams` and tags within them may represent images from multiple sources, including the OpenShift's internal registry, Other `ImageStreams` or external registries. [18]

When deploying an application to in OpenShift, the container images are referred to using the `ImageStreams`, in the same form

`ImageStreams` support scheduled importing of images from external registries. They also allow triggering new builds or deployments when the images change.

3.4.4 Builds

Openshift has a build system, which we can utilize to automate the building of the container images. Builds are define using an OpenShift object Build-Config. BuildConfig may be configured to initiate a new build automatically on various triggers, including webhook calls, pushes to GitHub or a change to an image in an ImageStream. BuildConfig can also be configured to run tests within a container from the built image before pushing it to the ImageStream [18]

3.4.5 Volumes

In OpenShift, persistent storage is represented by the *Persistent Volume (PV)* and *Persistent Volume Claim (PVC)* objects. The user defines a PVC describing the type of desired persistent storage. PVC includes, for example, size of the storage, storage class and access type. In response to PVC, the cluster administrator creates a PV that represents the actual storage. The user does not need knowledge of the actual underlying storage system. In a DeploymentConfig, user can use the created PVC as the volume source. The persistent storage represented by the PVC will be mounted inside the deployed container.[18]

3.4.6 ConfigMaps

For storing various kinds of configuration items, OpenShift's *ConfigMap* objects can be used. These are basically collections of key-value pairs that can be provided to a container using either of 3 different methods: 1) define the key-value pairs as environment variables inside the container, 2) use the values as a parameters on the container command, 3) Mount the key-value pairs as files inside the container.[18]

ConfigMaps are useful, for example, in passing single configuration values or whole configuration files to the deployed container.

3.4.7 Routes

The pods within the OpenShift cluster usually do not have public IP addresses. Instead the cluster has a single entry point. In order to route traffic in and out the cluster, OpenShift *Route objects* are used. These are an abstraction for configuration the underlying router. Route objects mainly define the following: 1) domain name 2) optional url path 3) Service to route

traffic to 4) SSL termination 5) Handling of insecure traffic 6) Certificate and private key.[18]

Defining a route makes all incoming route traffic matching the domain name and url path to be routed to the given service. If defined, the router also takes care of SSL termination and redirection of insecure http requests to https.

3.5 Docker Hub

Dockerhub[7] is a public registry for container images provided by Docker. Besides storing and sharing docker images, it Docker Hub can automatically build images from GitHub. Docker Hub also supports setting up webhooks, that are triggered when an image is updated.

Dockerhub hosts all the official Docker images, such as the base Debian, Ubuntu or Alpine distribution images. Docker Hub also hosts lots of docker images from external publishers.

In some cases, we can directly utilize images from external vendors. Or if suitable image is not available directly, we can build our own based on one of the official images or one of the images from external publishers. We can also host any of our own images, that can be made public, on Docker Hub. The automated builds on Docker Hub helps us with setting up continuous deployment.

3.6 GitHub

GitHub provides hosting of git repositories along with lots of features for software development.[9]

We can use GitHub to host code for all of our public applications and docker images. Integration with Docker Hub and support for triggering webhooks on code pushes helps us implementing continuous deployment.

3.7 GitLab

We also have a *GitLab*[10] service at our disposal, provided by the university. It has similar set of features to GitHub, but can be used to host private repositories. Private repositories are useful when we don't want to make the application code public or when the repository contains information that should not be made public. For example, configuration specific to our environment is something that we most likely do not want to be publicly visible.

Chapter 4

Implementation

This chapter tells how we implemented the migration. In Section 4.1, we introduce the old deployment and the architectural changes that were done in order to adapt the services for the new platform. In the Section section:containerization, we go through how we containerized the different types of services. In Section 4.3, the organization of the configuration repositories is introduced. In Section 4.4 we show how the building and deployment of the containers was automated. In Section 4.5, we implement an off-site backup system. In Section 4.6, we show how routing of traffic is handled in the OpenShift platform. In Section 4.7, the data is transferred from the old environment to volumes on OpenShift.

4.1 Architecture

In the old environment there were four servers: *frontend server*, *web server*, *app server* and *store server*. SSL termination caching and authentication were done on the frontend. Web had an Apache web server hosting regular web pages. App hosted mostly of Tomcat applications. Store was running a Fuseki database server with lots of datasets used by different applications.

4.1.1 Frontend

For caching, frontend utilized *Varnish Cache 3.0*[16]. The cache was configured using a C-like configuration language VCL. All network traffic for the services was going through this single cache server. The cache server was also responsible for redirects, rewriting URLs and even authentication. The configuration was quite complex and the logic was difficult to follow. Furthermore, there was some clear errors in the logic.

4.1.2 Web server

The *web server* was used mainly to host web services consisting of HTML pages, PHP and CGI scripts. It was also hosting a single Wordpress site along with its MySQL database.

4.1.3 App server

The *app server* was used to host variety of services, mostly Tomcat applications.

The server had a Tomcat instance with multiple application deployments. Most of the applications depended on a single or multiple datasets on the store server. The applications would make HTTP requests to the store server via the frontend server.

4.1.4 Store server

The *store server* was hosting a Fuseki SPARQL server[26]. The server had tens of different RDF datasets that are accessed over HTTP using the SPARQL Graph Store HTTP Protocol. The datasets were utilized by other services hosted on Web and App. For some datasets, the SPARQL endpoint was also made directly available to the internet, the data itself offered as a service.

Many of the SPARQL queries being made to the server are relatively heavy, often taking minutes to complete. This has been a major bottleneck for many services utilizing the data. The store also relies heavily on the frontend cache to speed up queries. Caching is effective since lots of queries are identical coming from the same web services. Therefore caching the SPARQL queries significantly improves the performance of the querying services.

4.1.5 Changes

We split the cache and its configuration that each service would have its own Varnish instance and only the relevant parts of the configuration. This made the individual cache configuration files simpler and easier to read. It also allowed scaling the cache on per service basis if needed. Since the cache is separate for each service, possibly badly behaving services will not affect the caching of other services.

This change unfortunately comes with a downside. In the original caching logic, there were some statements where the cache would be invalidated on multiple different services simultaneously. Namely this scenario comes up

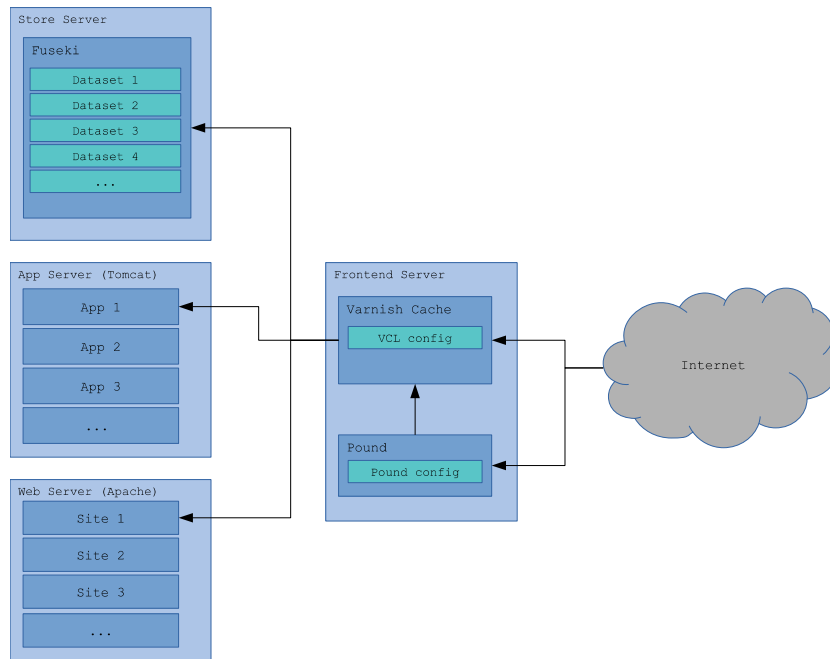


Figure 4.1: Old servers

when the data on store is updated - in such case the cache of possibly multiple services using the data also needs to be invalidated. Implementing such cache invalidation after splitting the cache to multiple instance might not be trivial.

We split the fuseki database server into multiple instances. We created separate fuseki instances for the bigger datasets and combined the smaller ones to a single instance. This should have mainly two benefits: 1) Make it possible to scale and allocate resources on per dataset basis. 2) Make it so that heavy queries on one dataset will not affect other datasets. This should somewhat alleviate the bottleneck that the heavy SPARQL queries were causing. Each fuseki instance would also get a dedicated Varnish instance.

The old on-premises deployment and the new OpenShift deployments are visualized in the figures 4.1 and 4.2.

4.2 Containerization

In this section, we show how the different types of services are containerized.

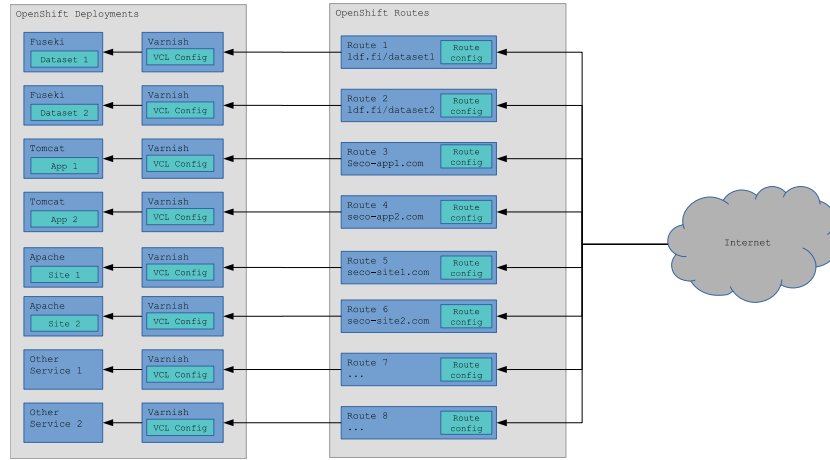


Figure 4.2: OpenShift deployment

4.2.1 Varnish Cache

We created a container image containing packages for *Varnish 5.0* as well as some additional modules for varnishes. The varnish is provided directly in the debian's package repository by the package 'varnish'. The extra modules are downloaded and built from a repositories in GitHub. Namely, modules built from the github repositories were `fastly/libvmod-urlcode`, `xcir/libvmod-parseform`[24] and `varnish/varnish-module`[32]. These modules provide various functions that are used in the VCL configuration for multiple different services.

The varnish container can be configured in few different ways depending on the situation.

Firstly, A simple caching behaviour can be configured by simply defining environment variables for the container. For example, `VARNISH_BACKEND_IP` and `VARNISH_BACKEND_PORT` variables can be used to define the backend to be cached. `VARNISH_DEFAULT_TTL` can be used to define the tome that backend responses are cached for. `VARNISH_MEM` can be defined to tell varnish the amount of memory to be used. By default, the varnish container serves the cached content at `http://<container>:80`.

In a more complicated scenario, a vcl configuration file can be mounted inside the container at `/etc/varnish/site.vcl` and `/etc/varnish/default.vcl`. The former way retains some default configuration included in the container image. The latter way can be used to override all of our defined default configuration.

We used the container in 2 different ways. For apache and tomcat based

services, we extended the varnish container by adding apache or tomcat installations to the image. This way, a single container image containing both, the backend service as well as the cache, was produced. Alternatively the varnish container could be deployed as a standalone in front of the service container. We used the latter especially with the Fuseki database services.

4.2.2 Apache websites

Majority of the services migrated were relatively simple web pages consisting of HTML/JavaScript pages, media files PHP and possibly some CGI programs. The web pages were hosted using the Apache web server.

For these services, we created a container image **apache-varnish**, which contained packages for varnish, Apache web server, PHP and few common PHP extensions.

Depending on the service, the **apache-varnish** container could be configured in a few different ways: 1) Using the included default configuration sufficient for some very simple sites 2) Modifying the default configuration using environment variables for some very common configurations 3) overriding the default Apache configuration by building a new image or mounting the configuration file or files. In most cases, the method 3 was used and a new image built based on **apache-varnish** that replaces the apache's vhost configuration at `/etc/apache/sites-enabled/000-default.conf`.

There are 2 different ways to deploy the actual content of the website (HTML, PHP, media etc.). The content could be build in to the container image or it could be stored on a volume and mounted inside the container.

Advantages in the first approach are in cleanliness and version control using a simple `COPY` command in a Dockerfile is easier than setting up volumes and configuring the mounting. Additionally, in this approach, the content is stored in the same repository with the Dockerfile and other configuration, meaning that the content is also versioned.

However building the content into the container image also means that possibly large files such as images or videos also gets stored and versioned in git resulting into large repositories which may lead to various problems. This way the content is also immutable, meaning that the content cannot be modified by the service. This is mainly a problem for services that need to modify files and store persistent data.

There was a single service that relied on CGI programs written in Python 2.7 and Python 3. We created a new container image based on the **apache-varnish**, that would add the packages for both Python versions and a Python virtual environments for executing the CGI programs. Some

modifications were needed in the CGI launch scripts in order for them to utilize the `venvs`.

Some of the sites were not compatible with PHP 7. Upgrading the code was not a part of this project. Therefore we created a slightly modified container image `apache-varnish:php-5`, that contained the older PHP version.

4.2.3 Wordpress sites

Amongst the migrated services, there were a couple of *Wordpress sites*[34]. *Wordpress* is a content management system written in PHP. In order to run, it needs a web server to host the application on, and SQL database. For Wordpress we created a new container image based on the `apache-varnish` image. For the database, we used the Red Hat's MySQL images[17].

The Dockerfile of Wordpress downloads and extracts the latest Wordpress release to a intermediate folder `/wordpress` inside the image. We defined environment variables for configuration that needs to be set in the Wordpress configuration file. The configuration includes, for example, the site URL and database credentials. A custom Wordpress configuration file was added with configuration items replaced by environment variables. An entrypoint was added that would copy the wordpress installation from `/wordpress` to the deployment directory `var/www/html`, if the wordpress installation does not already exist there. The entrypoint also uses `envsubst` to place the environment variables in the configuration file.

For the wordpress container a volume needs to be provided for persistent data at `/var/www/html`. Wordpress stores data such as user media and plugins inside this folder.

In the MySQL container, a volume needs to be provided for persistent data at `/var/lib/mysql/data`, which is where the MySQL data is stored at.

4.2.4 Tomcat applications

Usually Tomcat applications are usually located under a single folder:

```
<webapp>/<files>
<webapp>/WEB-INF/web.xml
<webapp>/WEB-INF/classes/
<webapp>/WEB-INF/lib/
```

However, in our case some applications also had some persistent data stored elsewhere in the system.

There are multiple ways to deploy applications on a Tomcat server. Tomcat automatically loads applications from the folder defined by an environment variable `CATALINA_BASE`. Tomcat includes a web application called "Tomcat Manager", which provides an HTTP API as well as web interface for managing the Tomcat server. [27]

Because a single container will be running only one Tomcat application, It is only required that a single Tomcat application is loaded at the container startup.

For some application, there was no source code available. In such cases, we simply copied the `webapp` folder from the old servers to the repository. We could create a simple Dockerfile, that copies the application on top of the generic Tomcat container image.

Where application source code was available, we included building the application from source in the the Dockerfile.

4.2.4.1 Java version

Some services that had no original sources available could not be run on newer Java versions. Therefore, we created a container image, `tomcat-varnish:tomcat-7`, which would instead contain older versions of Tomcat and Java: *Tomcat 7* and *Oracle JDK 6*. This made it possible, depending on the service, to simply copy the webapp, mount it inside the container or alternatively create a new image based on `tomcat-varnish:tomcat-7` with the webapp folder built-in. folder from the old server and run it as-is.

4.2.5 Fuseki databases

On a Fuseki 2 server [26], the data is stored in the folder indicated by the `FUSEKI_BASE` environment variable. Layout of this folder is as follows:

```
<FUSEKI_BASE>/
<FUSEKI_BASE>/config.ttl
<FUSEKI_BASE>/shiro.ini
<FUSEKI_BASE>/databases/
<FUSEKI_BASE>/databases/<dataset>/
<FUSEKI_BASE>/backups/
<FUSEKI_BASE>/configuration/
<FUSEKI_BASE>/configuration/<dataset-assembler>.ttl
<FUSEKI_BASE>/logs/
<FUSEKI_BASE>/system/
<FUSEKI_BASE>/system_files/
<FUSEKI_BASE>/templates/
```

We are mainly interested in `config.ttl`, `shiro.ini`, `.ttl` files under `configuration/`, and data stored under `databases/`. Rest of the files/folders are either not needed or are generated by the fuseki.

We created a fuseki container image that can be deployed and configured in a few different ways depending on the situation.

Without any configuration, the container runs fuseki with a preconfigured single dataset that is served over HTTP at the SPARQL endpoint `http://<host>/ds/sparql`. Using environment variables `ENABLE_DATA.WRITE`, `ENABLE.UPDATE`, and `ENABLE.UPLOAD`, other endpoints for updating the data can be enabled. `http://<host>/ds/sparql` endpoint can be used to update data using SPARQL Graph Store protocol. `http://<host>/ds/upload` can be used to upload data (non-SPARQL). `http://<host>/ds/update` can be used for SPARQL Update requests. The container also hosts and administrative interface at `http://<host>:3030`. In this case, a volume should be also mounted inside the container under `/fuseki-base/databases`, in order to persist the data between container redeployments/updates.

The folder `/fuseki-base` can also be directly copied over/mounted from another, existing fuseki installation. However it was discovered that using this method, the data might not be compatible as-is. For example, Lucene text index and SpatialLucene indexes from older fuseki versions are not compatible and needs to be regenerated in order for the indexes to work properly. Also the `.ttl` configuration files might need to be modified.

In case of read-only datasets, the data could be provided as `.ttl` files. A new container image based on the fuseki image can be created that loads in the data on build. In the Dockerfile of the new image, A script is run that uses a `tdbloader` tool to load the data to the fuseki. The built image therefore has `/fuseki-base` already populated with the data.

On the old server, the configuration was stored a bit differently. There was a single `fuseki-config.ttl` that contained configuration for all datasets. This is, while valid, an outdated way of configuring the datasets. In this migration, this central configuration file was split according to the newer layout to per-dataset assembler files.

4.3 Git repositories

For each of the services in the new environment, we created a Git repository for storing the service configuration. The repository has everything needed in deploying the service: 1) *Source code* or *Binaries*, 2) *Dockerfile*, 3) *OpenShift deployment configuration*, 3) *OpenShift route configuration*, 4) *README*. This structure is visualised in Figure 4.3

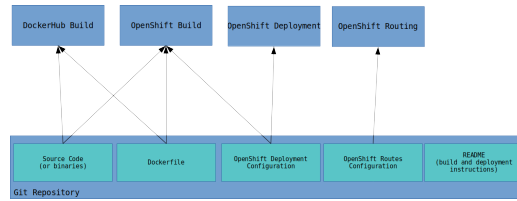


Figure 4.3: Git repository structure

This standard format ensures that the repository contains all the configuration items as well as instructions needed in building and deployment of a single service. In some cases the repositories are split in 2 parts, where the public container image build configuration is hosted on GitHub and OpenShift configuration specific to SeCo's deployment is stored in the private GitLab.

4.4 Automating Builds and Deployment

In the new environment, the aim was to make building and deploying the services completely automated. A new build of a container image should be initiated automatically by either 1) a *change to the source repository* from which the build is done or 2) *change to the base image* from which the container image is built. A new version of a container should also be automatically deployed when a new version of the container image is available. We automated the container builds and deployment solely using native features of GitLab, GitHub, DockerHub and OpenShift.

Builds initiated by the case can be implemented relatively easily on both build platforms: DockerHub and OpenShift. DockerHub features an integration with GitHub for automated builds. In OpenShift, a webhook based build triggers can be defined.

On DockerHub, we connect the image repository with a GitHub repository. Using the web interface, a set of build rules can be defined that link branches of a GitHub repository with tags in the DockerHub repository. Turning on "Autobuild" option on these rules sets the build so that it is automatically triggered whenever the corresponding branch in GitHub is updated. The user interface for configuring the build rules can be seen in Figure 4.4. It is also possible to enable "repository links" that also automatically trigger the builds when the base image defined by the FROM statement in the Dockerfile is updated. The repository links, however, do not work with the official images such as the ubuntu or debian base images.

On OpenShift, as a part of BuildConfig, we set up two kinds of build triggers in order to automate the builds: webhook triggers and image change triggers. These two types of triggers are shown in Figure 4.5.

Webhook build triggers gives an URL that, when requested, initiates a new build from the source repository. In the figure it is represented by 'triggers.generic' attribute. OpenShift creates an URL in the form `https://<openshift host>/apis/build.openshift.io/v1/namespaces/<project>/buildconfigs/<buildconfig>/webhooks/<secret>/generic`, as seen in Figure 4.6.

Image change build triggers initiate a new build whenever the base image is updated. In the figure it is represented by the 'triggers.imageChange' attribute. In the trigger the ImageStreamTag is defined that, when updated, initiates the new build. This works only with images within the OpenShift's internal registry. It is not possible, for example, directly point the trigger to an image in DockerHub. However there are 2 workarounds for this: 1) a scheduled import from DockerHub or other registries to the OpenShift's internal registry can be created. Triggers can then be based on the imported image. 2) DockerHub can also be set up to call the webhook build triggers in OpenShift. The method 1 was preferred, since the storage needed in the internal registry to import the additional images is not a problem and it works even with 3rd party repositories, such as the official distro images.

Both, GitLab and GitHub, allows configuring webhooks on push events to the repositories. The webhook can be pointed to the URL given to by OpenShift. This is done using the web interface of GitHub or GitLab. The user interfaces offered by GitHub and GitLab for configuring the webhooks can be seen respectively in the figures 4.10 and 4.11

For automating the deployment of newly built images, we configured triggers in the DeploymentConfig object in OpenShift as shown in Figure 4.8. In the trigger only needs the name of the ImageStreamTag that the built images are pushed to. OpenShift takes care of deploying the new image whenever the ImageStreamTag is updated.

The two main automation pipelines are visualized in Figure 4.9

4.5 Backups

The volume storage used in the OpenShift platform is redundant, but it does not feature actual backups or help in case of erroneous deletion of data. Therefore a backup solution for the data stored on the volumes is needed.

For this purpose, we set up a virtual machine that provides access over SSH to the university's NetApp based storage system. We also created an

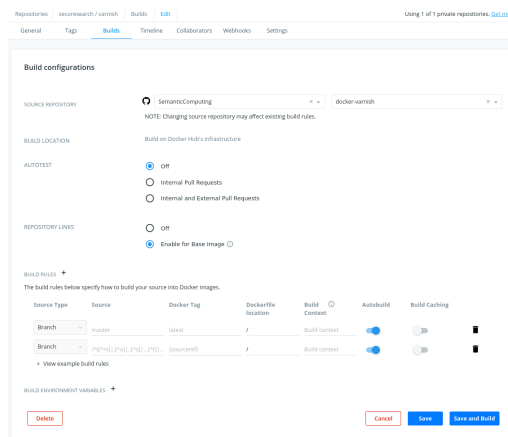


Figure 4.4: Setting up automated builds on DockerHub

rsync container that can push the data from the OpenShift volumes to the virtual machine

The virtual server is very simple Ubuntu installation. It has a single network drive mounted from the NetApp server. It has an OpenSSH server for SSH access, which is restricted to the OpenShift platform using a firewall. The system has a dedicated backup user with SSH access with key authentication.

The rsync container only has packages for ssh, and simply runs an rsync command at startup that synchronizes files from a given source folder to a given target folder. The source and target are defined by giving the container environment variables `RSYNC_SOURCE` and `RSYNC_TARGET`. These variables are fed as-is to the rsync command, so they can be local folders or remote locations. The ssh private key used to access the remote host and `known_hosts` file for identifying the remote target machine are mounted inside the rsync container. OpenShift ConfigMap resources are used for mounting these files.

For scheduling the backups, we are using separate OpenShift CronJob's for each volume to be backed up. The CronJob's configuration starts up the rsync container with the volume mounted to a specific folder inside the container, and sets up `RSYNC_SOURCE` and `RSYNC_TARGET` variables point to the mounted volume and the remote backup target machine respectively. The CronJob configuration also mounts the ssh key and `known_hosts` file from the ConfigMap.

The OpenShift volumes support RWX mode, meaning that they can be mounted inside multiple containers simultaneously. Therefore the backups can be run without interfering with the normal operation of the services using

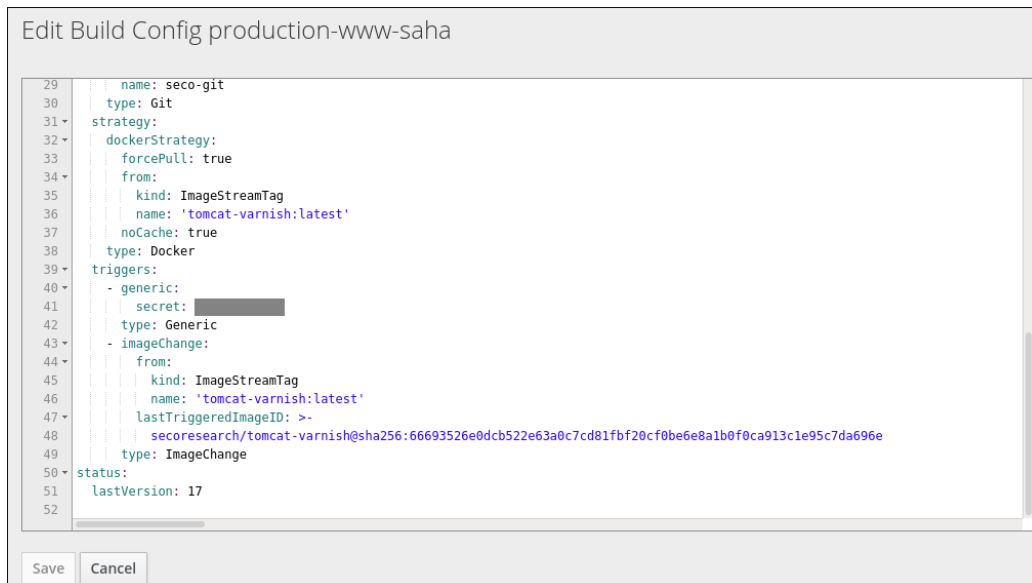


Figure 4.5: Openshift build triggers

the data.

We store all backup CronJob configurations in a single GitLab repository, along with a script that can be used to easily push the CronJob configurations to OpenShift.

4.6 Routing

The OpenShift cluster has a single IP address for ingress. The containers or pods are not directly visible to the internet and do not have separate public IP addresses. Therefore all DNS records to that IP address, and the OpenShift's router forwards the HTTP packages based on hostname and path to the backing pods and containers. The router does also SSL termination.

For configuring the routing of HTTP requests inside the cluster to the backing pod, we use OpenShift Routes. The route configuration needs to define at least the following: 1) Encryption (e.g. SSL termination or insecure) 2) Hostname to route 3) URL path to route 4) The OpenShift Service representing the backends (usually a Pod or multiple Pods). Route also handles the insecure traffic in case SSL encryption is used. The Route can be configured to e.g. redirect all HTTP traffic to the corresponding HTTPS URL. An example configuration of an OpenShift Route can be seen in the Figure 4.12

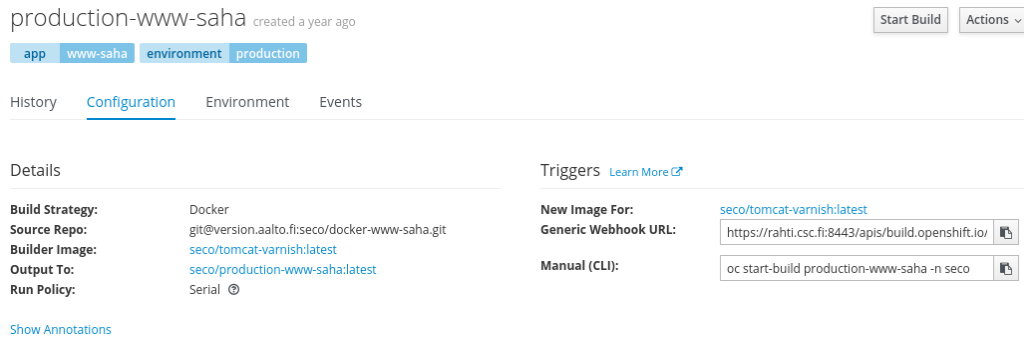


Figure 4.6: Openshift webhook URL

We defined separate routes for each domain name, or combination of domain name and path in case there is multiple services under a single domain name.

For majority of services, SSL encryption is enabled using Letsencrypt certificates. SSL termination is taken care by the OpenShift router as long as the certificate and private key is included in the Route configuration. An *OpenShift ACME controller*[30] is deployed within the project, which automatically acquires and renews certificates for the OpenShift Routes that have the annotation `kubernetes.io/tls-acme: "true"` defined.

Within each service's git repository, we include a script, that creates all the required routes for that service utilizing the OpenShift CLI. Having the scripts makes setting up the routes easier in case the service needs to be redeployed from scratch. The scripts also serve the purpose of storing the route configurations, domain names and URL paths associated with the specific service.

As with all other OpenShift objects, we label the routes with labels `texttapp:<service name>` and `environment:<environment>` to make it easy to identify which routes are associated with specific services.

4.7 Migrating Data

The only way for to access the OpenShift volumes is from within a running container. One way to transfer data to a container, is to run a container with the volume mounted and then use the OpenShift CLI's `'oc rsync'` command to transfer data from a local machine to the container. This works well for small amounts of data, but it was discovered that, at least on the OpenShift platform in question, the connections time out after some time.

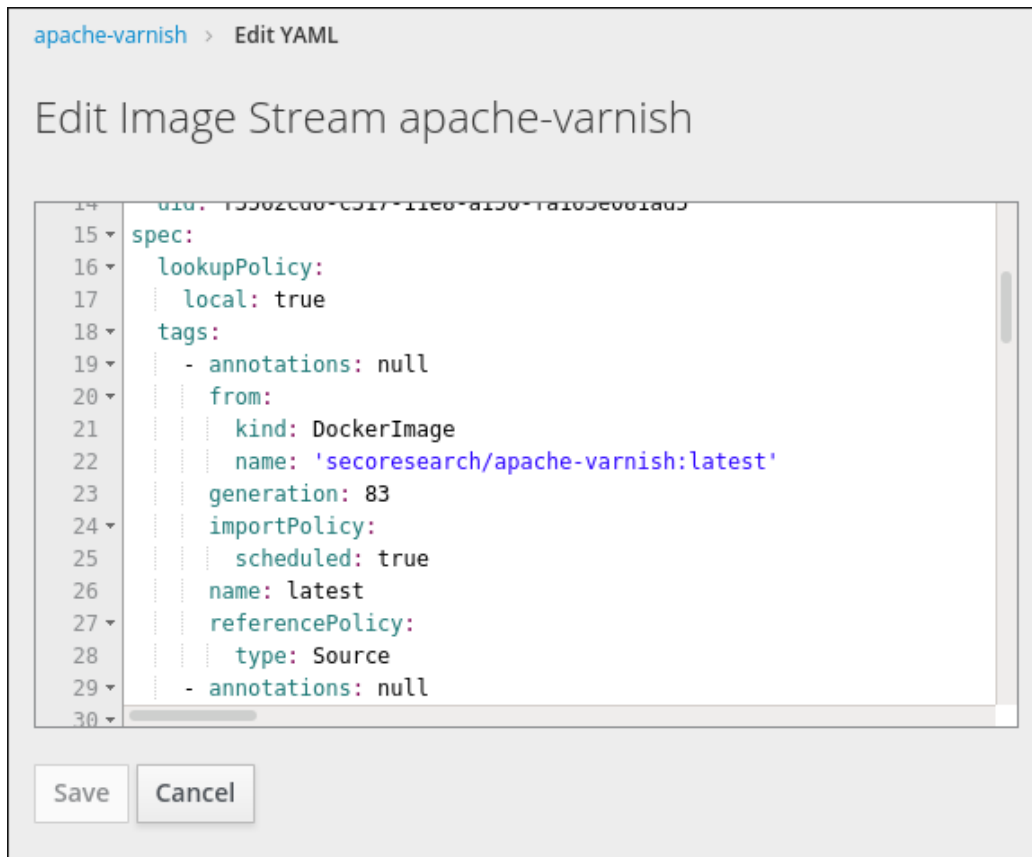


Figure 4.7: OpenShift scheduled import

Bigger transfers would time out before completion.

One way often recommended seems to be to run a container with a WebDAV server that could provide the containers filesystem over HTTP. On the OpenShift platform, It is not possible to route any other protocols besides HTTP to the containers, so using WebDAV would makes sense. However it was discovered that WebDAV lacks in some ways. For example, there does not seem to be a way to retain file timestamps when transferring files.

We resolved the data transfer problem by using a container that included **rsync**. We deployed the container to OpenShift, mounting the target volume the data needs to be transferred to and setting the container's command to e.g. 'sleep infinity'. A remote shell can then be the opened inside the container using the OpenShift CLI's 'oc rsh' command. rsync can be run from within the container in order to pull the data from the SSH server to the mounted volume. The remote shell connection, however, does timeout sim-

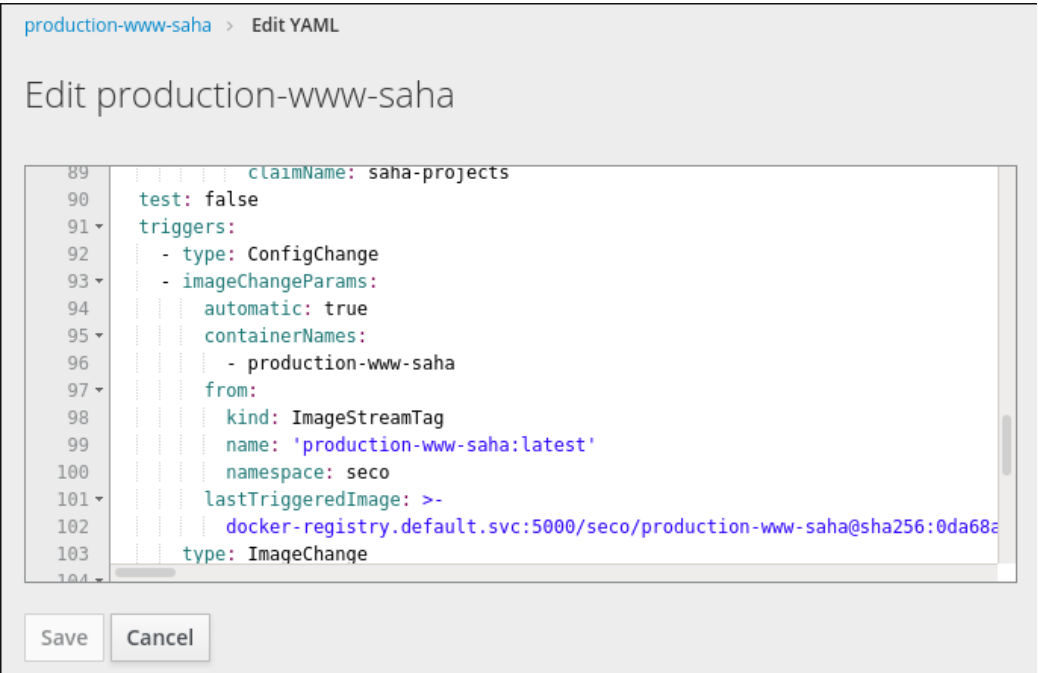


Figure 4.8: Openshift Deployment Trigger

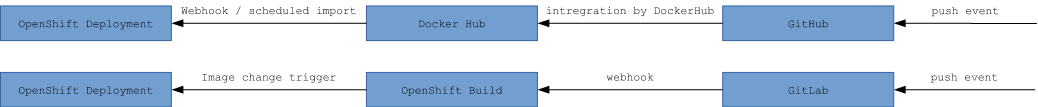


Figure 4.9: Automating builds

ilarily to the OpenShift CLI’s rsync command. This can be worked around this by running the rsync inside the container as a background job and then use ‘disown’ to prevent the rsync process from dying along with the shell when the connection times out.

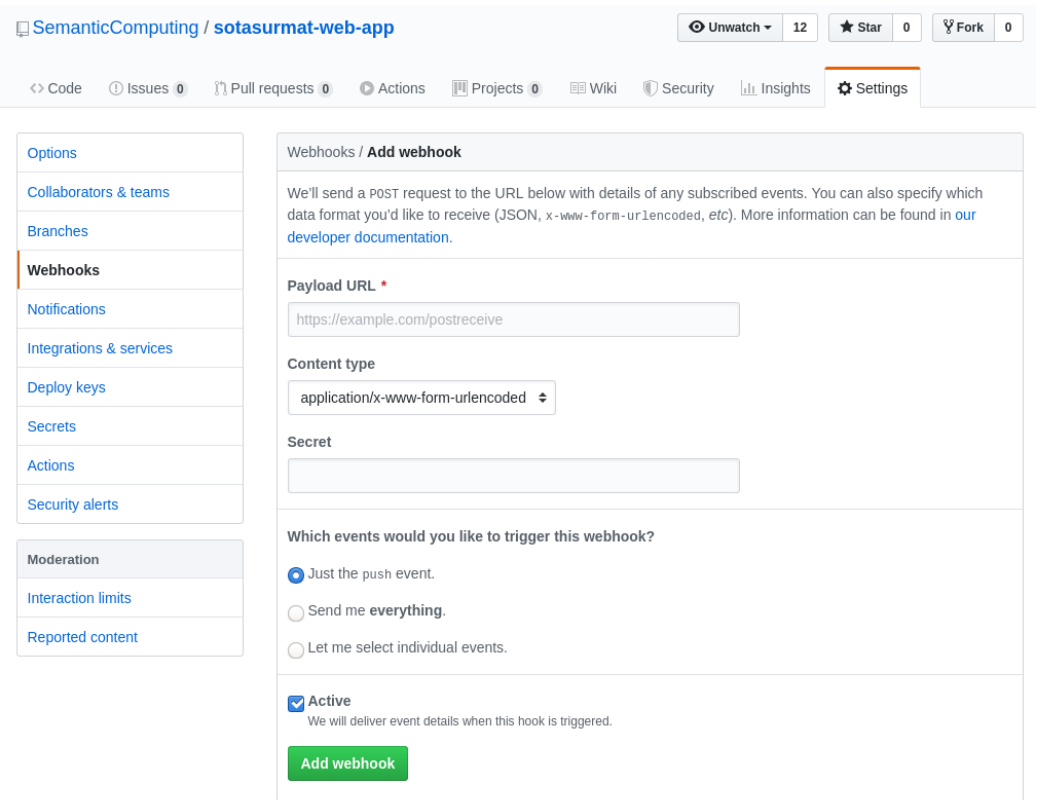


Figure 4.10: Setting up webhooks in GitHub

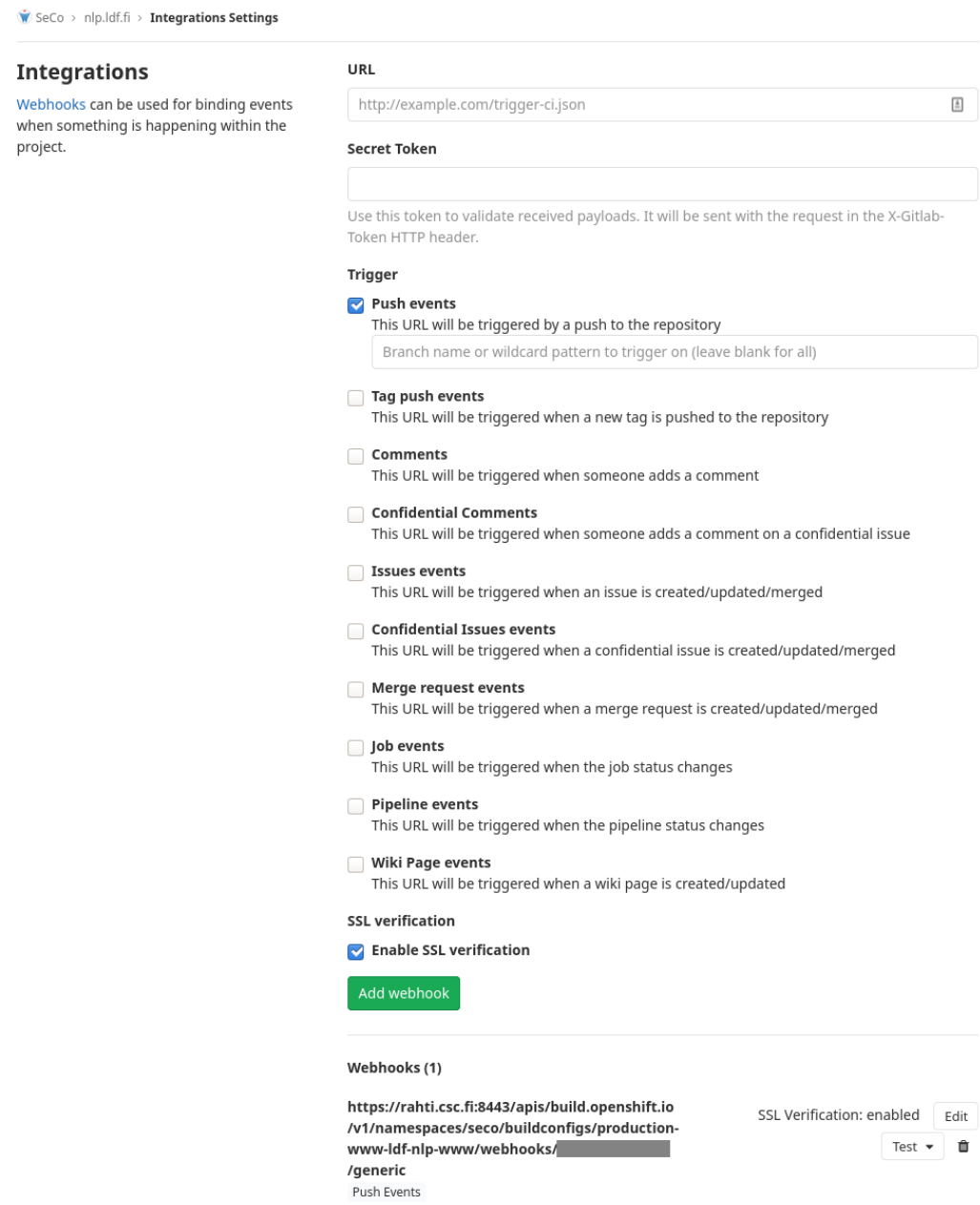


Figure 4.11: Setting up webhooks in GitLab

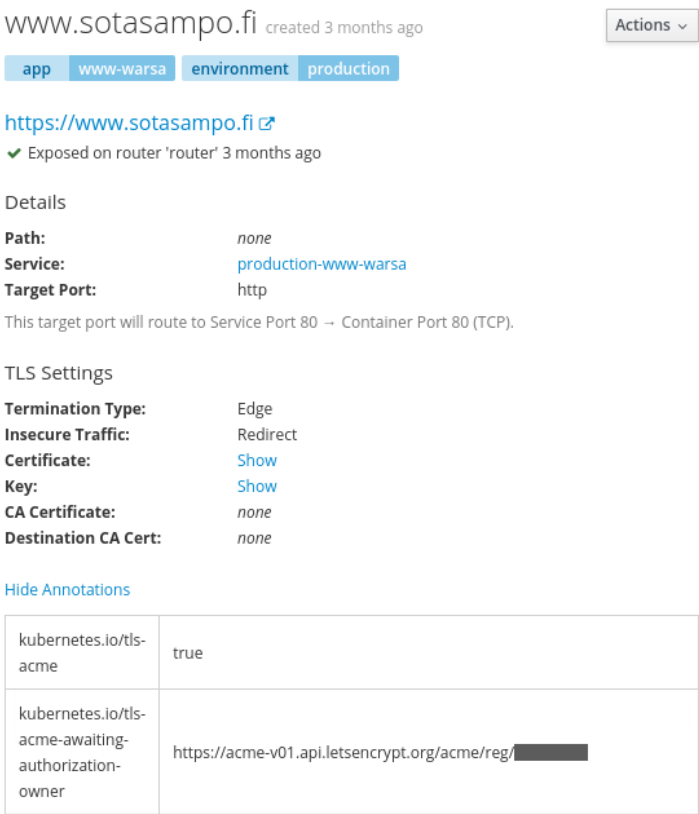


Figure 4.12: OpenShift Route Configuration

Chapter 5

Discussion

The migration project was completed successfully. All of the services and functions of the old servers containerized and moved to either the OpenShift platform or to a cloud virtual machine. The old servers have been decommissioned.

In this chapter we discuss about the outcomes of the migration project. We see to what was improved, what work is left as well as the drawbacks of moving into the cloud.

the first 6 sections in this chapter discuss the benefits of this migration. Splitting the server software to multiple instances, moving to PaaS and creating a standard format for configuration repositories improved maintainability of the system. Containerization and automation mitigated some of the security problems in the old system. Smaller services also lead to better scalability.

The last 8 sections in this chapter discusses the drawbacks and problems we found out as well as things that could still be improved in the new environment.

5.1 Splitting cache and database

We split up the large caching service to multiple instances. Each of the service would have their own cache service and configuration for it. While this lead to some code duplication, the individual configuration files are much shorter and easier to read and therefore less prone to errors. Some of the routing and redirect logic could be handed over to OpenShift routes, which further decreased the complexity of the individual cache configuration files.

Splitting up the monolithic database server to multiple instances reduced the risk of a single dataset consuming too much resources. It also allows

allocating more resources for the datasets that need them. This was not possible in the old environment.

5.2 Benefits from PaaS

The primary driver for this migration project was the aging server hardware. One of the main benefits of PaaS is that it frees the user from managing the computing resources[33][35]. Containerization technologies and OpenShift enables the usage of PaaS regardless of programming languages or software dependencies.

5.3 Standardized repositories

The included creating a dedicated git repository for each service. These repositories would include 1) a readme with a brief description of the service and it's deployment 2) Dockerfile if not using an off-the-shelf container image 3) An OpenShift template and parameters for the template

Having a brief readme for each service is useful in general. It gives a quick overview of what is included in the repository and how the service is deployed. The Dockerfile is not only used for building the container image, but it also works as a document about the dependencies of the service. The OpenShift template and it's parameters also work as a documentation about the deployment. From them one can deduce how the service is deployed, including 1) container image used 2) environment variables used 3) data locations indicated by the volumes mounted.

We believe that this standard format makes it easier to maintain the services in the future.

5.4 Service isolation

In the old server, multiple services were running on a single server. The server operating system along with many of the pieces of software were old and not supported anymore. This was a problem from the security point of view.

Containerization works as a layer of isolation between the different services. Even though there are multiple container running on the same host, the services are isolated from each other by the linux namespaces. Even though some services still require old versions of software in order to run, a

problem in a single service cannot compromise other services as it could've on the old servers.

5.5 Automated deployment

The automated deployments can be used to ensure that the software versions in the deployed images are always up-to-date. This should ensure that any security vulnerabilities due to outdated software/dependencies are mitigated as soon as possible.

Automation is an integral part of modern software development, CI/CD and DevOps practices. More from DevOps and Continuous Deployment can be read from [4] and [2]. Recent research on the topic includes a systematic review on CI/CD approaches, tools, challenges and practices, in 2017 by Shahin et al.[23]. Another example is an the master's thesis by Mustonen [15], where he evaluated benefits of Continuous Deployment based on interviews and a survey from software professionals.

5.6 Scalability

The migrated services were originally not designed specifically for containers or cloud. Therefore they do not benefit as much from the horizontal scaling of pods offered by the OpenShift platform. This would otherwise greatly improve scalability of the services. However we are not expecting the increase in the utilization of these services in the future so this should not be a problem.

Regardless, the platform allows us to set resource limits, namely cpu and memory, on per pod basis. This is useful as we can allocate resources specifically for the services that need them. Furthermore, badly behaving services cannot consume more resources than they are limited to, possibly affecting other services. This is an improvement compared to the old environment.

Getting benefits from scalability was not the driver for this project. Reengineering all of the services to take full advantage of the cloud would have not been feasible.

In more critical systems and applications handling lots of users and/or traffic, evaluating scalability and failure resistance is important. For example, S. Toimela[31], in his Master's Thesis, evaluated scalability and failure recovery in the context of telecommunication applications. K. Muhammad[12], also in his Master's thesis, looks at migration of monolithic payment application to containerized microservices architecture. In the thesis, improvements

on failure recovery and scaling times are evaluated.

5.7 Amount of work

The migration included adapting and moving 30 web services to the OpenShift cloud container platform.

Time used in the migration project was not tracked accurately. However we can approximate the effort the project took.

Most of the work was conducted by the author of this paper. We can approximate that the project lasted from January 2018 till end of January 2020. The author approximates that during this period, the time he spend on this project is comparable to 30%-40% of hours of a full-time employee.

I had experience from one similarly sized migration project, but no practical experience working with container technologies or cloud platforms. Therefore a significant amount of the work went to researching the relevant technologies and learning. Originally we thought that the migration would take 3-4 months. However the services had to be migrated one by one and more often than not, something unexpected would come up. The project ended up lasting multiple times longer than we originally thought.

Not doing the reengineering for cloud and migrating the programs and data as-is to a new virtual machines would have likely taken much less time.

5.8 Code duplication

One of the drawbacks of a microservice architecture is duplication of code and functionality[20]. Even though a microservice architecture was not intentionally being implemented in this project, some of the drawbacks of a microservice architecture was realized when splitting some of the service. For example, each service having their own cache meant that some of the cache configuration was written multiple times for different services. The configuration did have some service specific variation, so the same configuration could not be used for all of the services either.

This was addressed to some degree by including the common parts of the configuration to the cache base image and introducing another, service specific configuration on top of that. However this approach adds coupling and configuration split to multiple files is harder to read and understand.

5.9 Multiple processes per container

In the beginning of the project, a new container image was built for each service. This meant that, for example, for Apache based web sites, we would need to run two services within a container: The Apache web server and Varnish Cache in order to implement the logic from the previous cache on the frontend server. The reasoning behind this was that it would be simpler to create write only a single Dockerfile for a service. However it is generally advised that a container should not include more than a one service. We too, did discover that multiple processes within a container comes with some complications.

Firstly, we need to execute the processes on the background. This makes signal handling complicated. By default, whenever a container is stopped by the orchestrator, the process id 0 is sent a **SIGTERM** so it can terminate gracefully. However, this means that the signal, depending on the implementation, does not necessarily propagate to the processes on the background. Therefore, in the case of an Apache web site, the Apache server or the Varnish Cache might not be terminated gracefully. The signals should therefore be explicitly propagated to the child processes

Secondly, logging gets more complicated. Usually, the containers simply log to stdout. So in order to get logs from the both processes, we need to somehow concatenate logs from both of them to the stdout. We would work around this by first forwarding the logs from each process to separate files. Then would execute `tail -f <log files>` as the foreground process. This works to some degree, but we found out that in some cases this would result in problems like a lot of empty lines being printed in the logs.

The overhead introduced by containerizing a process is insignificant in regards of performance so performance is not a reason to include multiple processes within a container.

Separating the processes to their own containers remains as a future improvement.

5.10 Making containers more generic

During the project, for many of the services, configuration and data was built-in to the container image. This meant that a new container image was created for each of the services. This seemed reasonable as we could build in the configuration and in some cases data into the container. This would allow deploying the image as is without further configuration. This is not necessarily a bad practice. For example, documentation of the official nginx

image[8] mentions this as the 'cleaner way'

However in some cases it might be better to create more generic images and mount the configuration items and data utilizing OpenShift ConfigMaps and Volumes. This means that less images are built and stored. The images built also would not contain any sensitive information as no specific configuration or data is included. However this approach requires extra platform specific configuration. In OpenShift this means uploading the configuration ro data and writing configuration for mounts.

5.11 Storage problems

During the migration project, there were multiple occasions where the volume mounts would fail on OpenShift. The problems were likely caused by capacity problems in the underlying storage system. Towards the end of the migration the storage was more stable.

5.12 Monitoring

The OpenShift's monitoring interface a reasonable summary of what is happening within the OpenShift project. It also shows any errors and warnings from builds and deployments. However it does not automatically inform, for example, via email if something goes wrong.

Also some of the builds are done in DockerHub. So there are multiple platforms to be monitored. Therefore it would be beneficial to have a single monitoring solution that would gather the information from these multiple sources and inform the owner of the services in case there is an error. This remains as a future work.

5.13 Testing

Currently there isn't much automated tests done. In most cases, the only way to test after doing changes to a service, is to deploy the container locally. Even then, it is not always possible to thoroughly test the services, as they might rely on data or integrations not available on a local development machine.

Dockerhub and OpenShift both have features that enable testing even without a dedicated CI/CD solution such as Jenkins or Buildbot. DockerHub lets us set up automated tests using a docker-compose file included in the source repository an image is built from. The docker-compose file

is deployed before the built image is pushed the DockerHub repository. If the deployment defined by the docker-compose file fails, the build will result in an error instead. In the docker-compose file, it is also possible to define dependent services allowing us, to some degree, run even integration tests within DockerHub

Automated testing is an important part part of and modern software development and DevOps practices[4][2].

5.14 Cost Optimization

One characteristic of cloud is that usage is billed by metered usage. This project was not limited by resources and optimization of resource usage was not a focus. However optimizing the resource requests and limits of the deployments and setting proper limits is something that should be addressed in the future. Especially finding proper memory limits for the Java based applications, since Java's memory management doesn't work well in containerized environment and it needs explicit limits to be set.

Chapter 6

Conclusions

This work focused on a migration of existing services and showed how containerization can, in addition to developing new software, also be used in modernization efforts of existing software.

We migrated the web services of the SeCo research group from an on-premises servers to OpenShift. The migrated services included simple websites and tomcat applications and databases. Some of the services were deployments of off-the-shelf applications. All of the applications were containerized and they were mainly deployed on an OpenShift container platform.

The project demonstrated how the different application types can be containerized. The project showed the problems that were encountered and how they can be handled. Each application type comes with its own considerations when moving to a containerized deployment.

We talked about how the containerization technologies improved the quality, especially maintainability and security of the migrated services. Downsides of the migration were brought up including the amount of work included in learning the new technologies and reengineering existing services as well as new complexities introduced by moving towards microservice like architecture. Various improvements for the future were also discussed.

Containerization has only recently seen a big increase in popularity due to enabling technologies such as Docker, Kubernetes and support from public cloud providers. Increasing amount of new software is developed and deployed using container technologies so it is important to keep researching the potentials as well as draw-backs of them.

Bibliography

- [1] Docker. webpage. <https://www.docker.com/>. Accessed 15 May 2019.
- [2] Amazon Web Services, Inc. What is DevOps? webpage, 2020. <https://aws.amazon.com/devops/what-is-devops/>. Accessed 16 May 2020.
- [3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010. doi: 10.1145/1721654.1721672.
- [4] Atlassian. What is DevOps? webpage, 2020. <https://www.atlassian.com/devops>. Accessed 16 May 2020.
- [5] Docker Inc. Docker Documentation. webpage, 2019. <https://docs.docker.com/>. Accessed 15 May 2019.
- [6] Docker Inc. What is a Container. webpage, 2020. <https://www.docker.com/resources/what-container>. Accessed 29 Mar 2020.
- [7] Docker Inc. DockerHub. webpage, 2020. <https://hub.docker.com/>. Accessed 29 Mar 2020.
- [8] Docker Official Images. nginx. webpage, 2020. https://hub.docker.com/_/nginx. Accessed 29 Mar 2020.
- [9] GitHub Inc. GitHub. webpage, 2020. <https://github.com/>. Accessed 29 Mar 2020.
- [10] GitLab. GitLab. webpage, 2020. <https://github.com/>. Accessed 29 Mar 2020.
- [11] Ravi Khadka, Belfrit V Batlajery, Amir M Saeidi, Slinger Jansen, and Jurriaan Hage. How do professionals perceive legacy systems

- and software modernization? In *Proceedings of the 36th International Conference on Software Engineering*, pages 36–47, 2014. doi: 10.1145/2568225.2568318.
- [12] Muhammad Khan. Scalable invoice-based b2b payments with microservices. G2 pro gradu, diplomityö, 2020. URL <http://urn.fi/URN:NBN:fi:aalto-202001261876>.
- [13] Kubernetes. Understanding Kubernetes Objects. webpage, 2020. <https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>. Accessed 3 Apr 2020.
- [14] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.
- [15] Aleksi Mustonen. Ways to improve continuous deployment processes. G2 pro gradu, diplomityö, 2020-03-16. URL <http://urn.fi/URN:NBN:fi:aalto-202003222623>.
- [16] Poul-Henning Kamp. Varnish HTTP Cache. webpage, 2020. <https://varnish-cache.org/>. Accessed 29 Mar 2020.
- [17] Red Hat, Inc. Using Red Hat Software Collections Container Images. webpage, 2020. https://access.redhat.com/documentation/en-us/red_hat_software_collections/2/html-single/using_red_hat_software_collections_container_images/index. Accessed 29 Mar 2020.
- [18] RedHat. OKD 3.11 Documentation, . <https://docs.okd.io/3.11/welcome/index.html>. Accessed 29 Mar 2020.
- [19] RedHat. OpenShift, . <https://www.openshift.com/> Accessed 15 Apr 2020.
- [20] Mark Richards. *Microservices vs. service-oriented architecture*. O’Reilly Media, 2015.
- [21] John W Rittinghouse and James F Ransome. *Cloud computing: implementation, management, and security*. CRC press, 2016.
- [22] Robert C Seacord, Daniel Plakosh, and Grace A Lewis. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

- [23] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.
- [24] Shohei Tanaka. vmod-parseform. webpage, 2020. <https://github.com/xcir/libvmod-parseform>. Accessed 29 Mar 2020.
- [25] Ian Sommerville. *Software Engineering GE*. Pearson Australia Pty Limited, 2016.
- [26] The Apache Software Foundation. Apache Jena Fuseki. webpage, 2020. <https://jena.apache.org/documentation/fuseki2/index.html>. Accessed 29 Mar 2020.
- [27] The Apache Software Foundation. Apache Tomcat 8 Documentation. webpage, 2020. <https://tomcat.apache.org/tomcat-8.5-doc/index.html>. Accessed 29 Mar 2020.
- [28] The Linux man-pages project. cgroups, . Copy of text available at <http://man7.org/linux/man-pages/man7/cgroups.7.html>. Accessed 29 Mar 2020.
- [29] The Linux man-pages project. namespaces, . Copy of text available at <http://man7.org/linux/man-pages/man7/namespaces.7.html>. Accessed 29 Mar 2020.
- [30] tnozicka. openshift-acme. webpage, 2020. <https://github.com/tnozicka/openshift-acme>. Accessed 29 Mar 2020.
- [31] Samu Toimela. Containerization of telco cloud applications; ohjelmistokonttien hyödyntäminen pilvipohjaisen mobiiliverkon sovelluksissa. G2 pro gradu, diplomityö, 2017. URL <http://urn.fi/URN:NBN:fi:aalto-201706135434>.
- [32] Varnish Software. varnish-modules. webpage, 2020. <https://github.com/varnish/varnish-modules>. Accessed 29 Mar 2020.
- [33] Quang Hieu Vu and Rasool Asal. Legacy application migration to the cloud: Practicability and methodology. In *2012 IEEE Eighth World Congress on Services*, pages 270–277. IEEE, 2012.
- [34] WordPress. WordPress. webpage, 2020. <https://wordpress.org>. Accessed 29 Mar 2020.

- [35] Jun-Feng Zhao and Jian-Tao Zhou. Strategies and methods for cloud migration. *international Journal of Automation and Computing*, 11(2): 143–152, 2014. doi: 10.1007/s11633-014-0776-7.