Bachelor's thesis

Business Information Technology

Information Systems

2016

Miina Koskinen

MICROSERVICES AND CONTAINERS

– Benefits and Best Practices

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Business Information Technology | Information Systems

2016 | 39

Anne Jumppanen

Miina Koskinen

MICROSERVICES AND CONTAINERS

Benefits and Best Practices

Microservices architecture and containers are fairly new technologies used in software development in cloud environments. They have become more popular and widely used the recent years. Microservices architecture is replacing the old fashioned way of developing a software system as a monolithic architecture. Microservices architecture is claimed to solve many issues that comes with a large scale monolithic systems.

The goal of the thesis was to evaluate the benefits and best practices of microservices architecture and containers based on literature, expert interview and a case study. The case study was Ericsson´s OneCloud project and especially the development of the OneCloud API.

During the research it was discovered that the key benefits of microservices architecture are; ease of maintenance, independent lifecycle of the microservices, independent scaling of the microservices and finally technology heterogeneity; in other words no dependence on one technology stack. It was also discovered that with development of microservices comes issues that don´t exist when developing a monolithic system. However these issues are avoidable but demand new development methods and tools.

The best practices that were found during the research were more of a general guidelines rather than detailed instructions on best tools and procedures. These practices can be used as a starting point when beginning a development with microservices and containers.

In conclusion the results found only scratch the surface of microservices and containers. It would demand more studying and exploration to better understand the benefits of microservices architecture and how to reach those benefits.

KEYWORDS:

microservice, software engineering, cloud services

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittely | Tietojärjestelmät

2016 | 39

Anne Jumppanen

Miina Koskinen

MIKROPALVELUT JA KONTIT

Hyödyt ja parhaat käytännöt

Mikropalveluarkkitehtuuri ja kontit ovat verrattain uutta teknologiaa, jota käytetään pilvipalvelupohjaisessa ohjelmistotuotannossa. Niiden käyttö on lisääntynyt viime vuosina. Mikropalveluarkkitehtuuri korvaa vanhanaikaisen tavan kehittää järjestelmiä monoliittisessa arkkitehtuurimallissa. Mikropalveluarkkitehtuurin on sanottu ratkaisevan monia monoliittiseen arkkitehtuurimalliin liittyviä ongelmia.

Tämän opinnäytetyön tavoitteena oli selvittää mikropalveluarkkitehtuurin ja konttien hyötyjä ja parhaimpia käytäntöjä kirjallisuuden, asiantuntijahaastattelun sekä tapaustutkimuksen perusteella. Esimerkkitapauksena toimi Ericssonin OneCloud projekti ja erityisesti sen ohjelmointirajapinnan (OneCloud API) kehitys.

Tutkimuksen aikana selvisi, että tärkeimmät hyödyt mikropalveluarkkitehtuurista ovat; vaivaton ylläpito, mikropalveluiden itsenäinen elinkaari sekä itsenäinen skaalaus ja lopuksi teknologinen riippumattomuus. Tutkimuksen aikana selvisi myös, että mikropalveluja kehittäessä tulee vastaan uudenlaisia ongelmia, joita ei esiinny monoliittisissa järjestelmissä. Nämä ongelmat ovat kuitenkin vältettävissä, ne vain vaativat uudenlaisia ohjelmistotuotantometodeja ja -työkaluja

Parhaat käytännöt, joita tutkimuksen aikana löydettiin, olivat ennemminkin yleisiä toimintaperiaatteita, eikä niinkään yksityiskohtaisia ohjeita parhaista työkaluista ja menettelyistä. Näitä käytäntöjä voidaan käyttää lähtökohtana kun aloitetaan mikropalveluiden ja konttien kanssa työskentely.

Lopputulemana voidaan sanoa, että tämä opinnäytetyö jää varsin yleiselle tasolle. Vaatisi laajempaa tutkimista, jotta selviäisi mikropalveluiden ja konttien hyödyt laajemmin sekä miten parhaiten hyödyt saataisiin käyttöön.

AVAINSANAT:

mikropalvelu, ohjelmistotuotanto, pilvipalvelut

CONTENT

LIST OF ABBREVIATIONS

API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Deployment
CPU	Central Processing Unit
DB	Database
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
I/O	Input/Output
JAR	Java Archive package file
MongoDB	Open source document-oriented database
MySQL	Open source database management system
OS	Operating System
QA	Quality Assurance
RESTFull API	Respresentational State Transfer Application Programming Interface
SOA	Service-oriented Architecture
TDD	Test-driven development
VM	Virtual machine
WAR	Web Application Archive file

1 INTRODUCTION

Microservices has been the hot topic in software development for couple of years now. Since the emergence of the open source Docker containers, which were released on March 2013 (Avram 2013), microservices have become even more popular. Few companies have already moved their software systems from monolithic architectures to architecture based on microservices. Those companies include for example Netflix, SoundCloud, eBay, Amazon and Google (Fowler 2016).

Microservices are part of microservices architecture which "*.. is a cloud-native architecture that aims to realize software systems as a package of small services"* (Balalaie & Heydarnoori 2016). Microservices architecture promises a lot; adaptability to technological changes, reduced time to market, better development team structuring around services (Balalaie & Heydarnoori 2016), better scalability, (Balalaie, Heydarnoori & Jamshidi 2015) improved fault isolation and more understandable code (Badola 2015). This thesis examines those promises, can development team truly reach all those benefits and if so, how it should be done?

Ericsson is also taking steps towards microservices. Ericsson is a Swedish multinational corporation that provides communication technology and services. Ericsson has strong history in telecommunications, approximately 40 percent of worlds mobile traffic runs through networks supplied by Ericsson (Ericsson 2016). Today Ericsson is moving towards new areas such as IT and cloud.

The author of this thesis has been introduced to microservices while working in Ericsson's OneCloud project which aims to develop a public hybrid cloud. The author has been working in the API development team developing an API which communicates with all the different components of the system, such as frontend, OpenStack, Apcera and CleverSafe. Microservices will be used in the future in the development of the OneCloud API and with this thesis the author wants to investigate what the best practices for using microservices are.

Although the OneCloud API isn´t yet following a microservices architecture it is developed with the mindset that it will be later on migrated to microservices. Containers are already used in the development. That is why the author uses the OneCloud API as

a case study for her thesis. This thesis can be used as a starting point for the development with microservices.

The questions what the author tryes to answer in this thesis are; *What are the benefits of microservices architecture? And what are the best practices for using microservices and containers?*

To answer these questions the author drew from her own experiences in Ericsson's OneCloud project. She also interviewed David Kuridza, a software developer from a Slovenian company called 3fs, who is familiar with microservices. She asked his experiences; success stories and failures. Furthermore the author looked into the large number of articles and literature what has been written about microservices.

2 MICROSERVICES AND CONTAINERS EXPLAINED

2.1 Background

Service-oriented architecture (SOA) is the predecessor of microservices architecture. This architectural style was adopted by numerous IT companies in the mid-2000´s. (Richards 2016, 9) SOA, as well as microservices architecture, aims to overcome the challenges with large monolithic architectures (Newman 2015, 28). In Fowlers opinion SOA hasn´t been well defined and the concept is too ambiguous (Fowler 2015) Newman also shares this opinion: *"The number of things that go unsaid is where many of the pitfalls associated with SOA originate. The microservice approach has emerged from real-world use, taking our better understanding of systems and architecture to do SOA well."* In Newmans opinion microservices is a specific approach for SOA and not a new architectural style on its own. (Newman 2015, 28) More on the technical side of SOA and the differences between SOA and microservices architecture on the next chapter *2.2 Microservices Architecture.*

Microservices is an important part of new software development concepts such as *Continuous Deployment* and *DevOps.* Continuous Deployment and its predecessors *Continuous Integration* and *Continuous Delivery* all aim for the same thing; to optimize the software delivery process and make the deployments as smooth as possible. Optimizing the process results in more frequent system releases. By releasing more frequently software developers get rapid feedback from users and know if they are building the right kind of product. The risks of release reduces and the developers get real process. (ThoughtWorks 2012)

The difference between Continuous Integration, Continuous Delivery and Continuous Deployment is the level of automation used and the level of production readiness after successful run through of the pipeline. In Continuous Deployments case it is assumed that "*every successful run of the pipeline can be deployed to production."* (Farcic 2016, 18) The pipeline refers to the workflow which the code goes through. Common Continuous Deployment pipeline includes following steps:

- **Continuous Integration tool** (for example Jenkins)**:** detects changes in the code repository, checks out the code and continues with the rest of the steps of the pipeline.
- **Static analysis:** performed without executing the code.
- **Pre-deployment testing:** tests that do not require code to be deployed.
- **Packaging:** code is compiled and possibly container created with all dependencies (libraries, runtime, application server etc.)
- **Deployment to Quality Assessment:** the package or the container is deployed to one or more test environments.
- **Post-deployment testing:** tests that require deployment.
- **Deployment to production:** the package or the container is deployed to the production environment.
- **Post-deployment testing:** tests that verify that the service or the application is integrated with the rest of the system.
- **Proxy:** proxy service is reconfigured to point to the new release. (Farcic 2016, 21)

In case there is a failure in any of the steps in the pipeline a failure notification is sent and the pipeline process is aborted. When a failure happens, fixing the problem becomes priority number one. If the failure notification is ignored following executions of the pipeline will fail as well and the Continuous Delivery process will begin to lose its purpose. (Farcic 2016, 21 & 12)

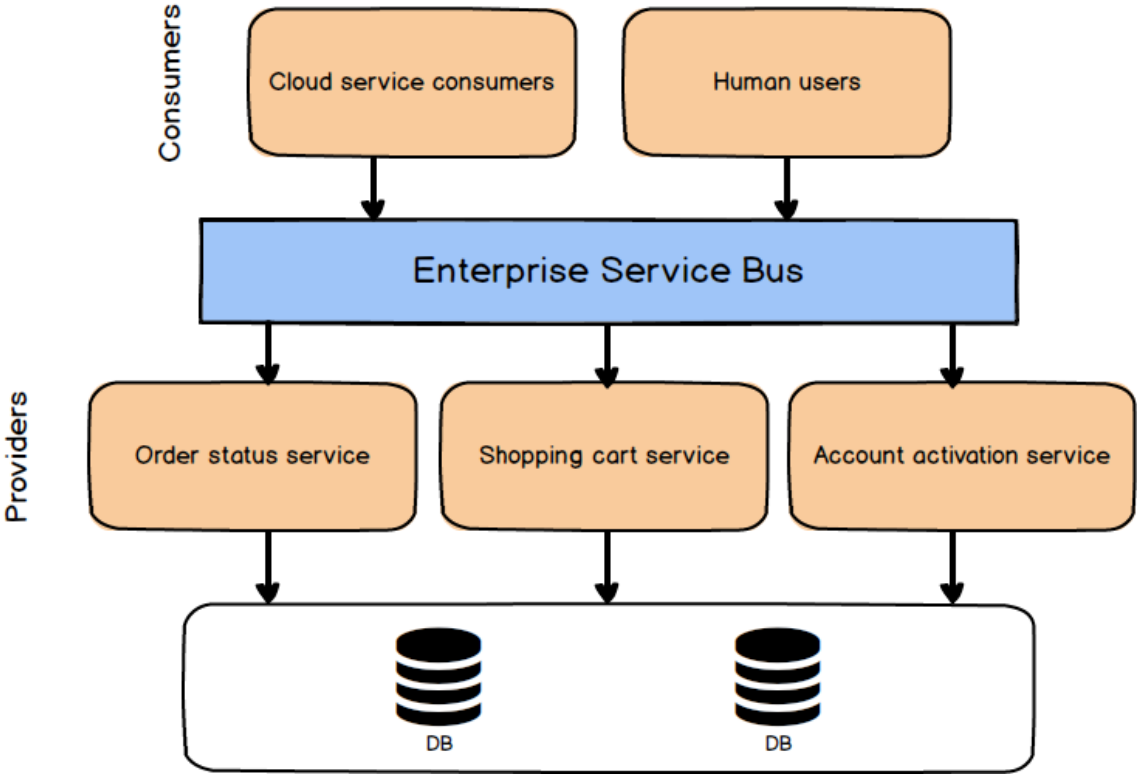
Where Continuous Deployment is about technical implementations, DevOps on the other hand is more of a cultural movement in software development. DevOps aims to remove deployment, operations and maintenance from separate silos and encourage collaboration between development and operations. With DevOps there´s an attitude of shared responsibility of the hole service lifecycle which will lead to better and more easily maintainable products. DevOps supports autonomous teams. Developers and operations staff need to be able to make decisions and apply changes without a heavy decision making process. With DevOps culture it becomes easier to put new code in production. With this change teams need to value building quality into the development process. Teams need to consider cross-functional concerns such as performance and security. Feedback is also an important aspect, both feedback from team collaboration and from the system itself. Final important character of DevOps is automation. Tasks such as testing, configuration and deployment are automated and people cam focus on

other valuable activities. Automation also reduces the chance of human error. (Wilsenach 2015)

Microservices play an important part on both of these new software development concepts; Continuous Development and DevOps.

2.2 Microservices Architecture

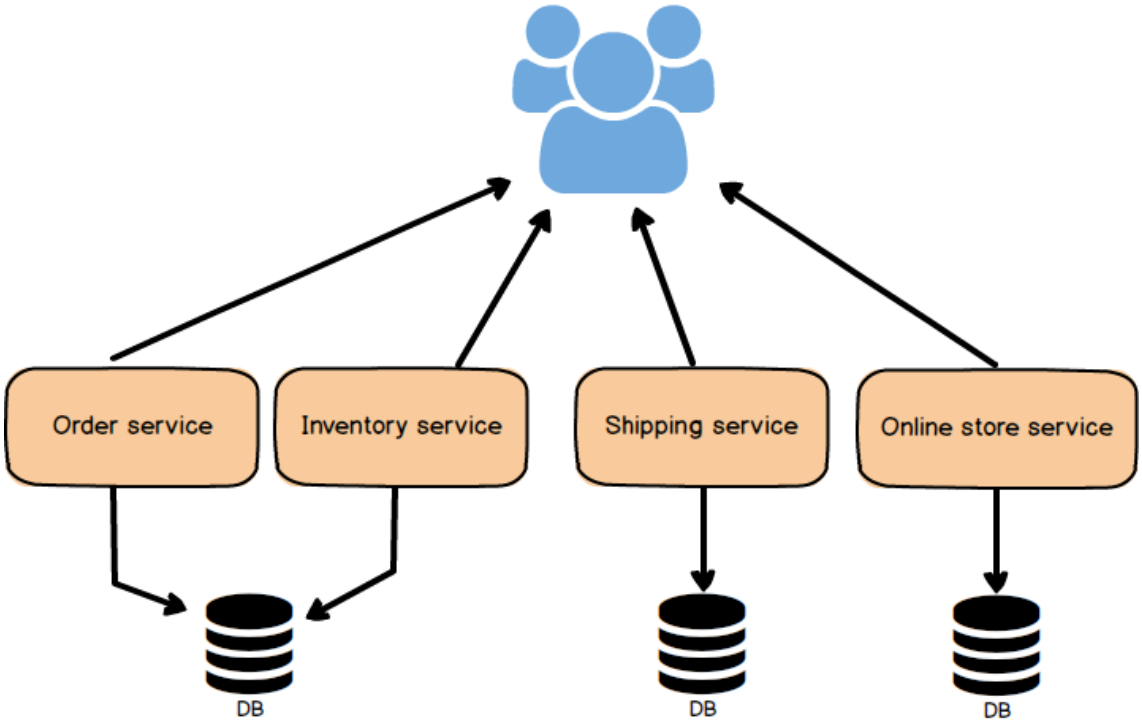
As previously mentioned some developers see microservices architecture as some form of service-oriented architecture (SOA). These two architectural styles nevertheless have lots of differences. In SOA, application components provide services to other components. These components communicate via communications protocol over a network. The communication can be either simple data passing or more complex coordination between different services. Services are responsible for small functions such as registering a new user. (Miri 2016) In below picture is an example of a basic SOA system. However it´s important to note that SOA has been designed and implemented in various styles and this is only one approach.



Picture 1. Service-oriented architecture (SOA) (Miri 2016)

There are two main layers in SOA, the consumer and the provider layer. In the consumer layer consumers, such as users, other services or third parties, interact with the SOA. The provider layer contains all the services of the SOA. (Miri 2016) These two layers communicate via Enterprise Service Bus (ESB) which is an integration architecture that enables communication via a common communication bus with several point-to-point connections between the provider and the consumer layer (Kress etc. 2013). ESB decouples the services, allowing them to communicate without dependence or knowledge of each other. In addition in SOA there´s also a data storage which is shared within all the services.

In microservices architecture applications are composed of small independent processes (microservices) which communicate through language-agnostic APIs. Each microservice has its own database or the database is shared between few microservices. Below is a picture of a basic microservices architecture system.



Picture 2. Microservices architecture (Miri 2016)

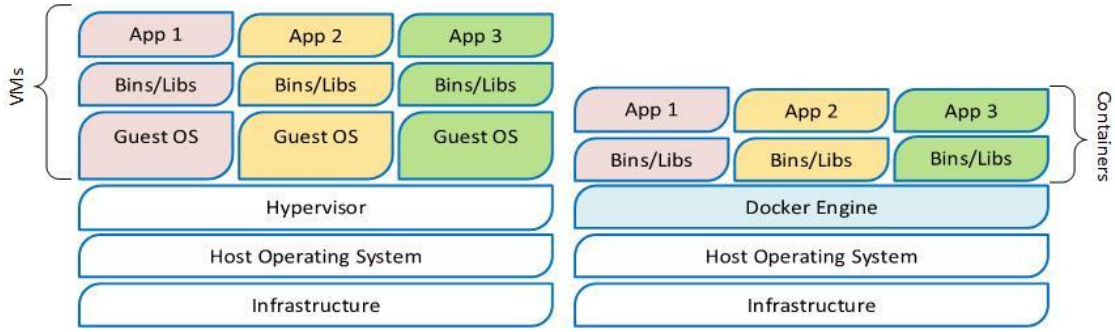
SOA and microservices architecture differ in multiple ways. In microservices architecture services can operate and be deployed independently which is not possible in SOA. In SOA the ESB can become the bottleneck since every service is communicating through it. The data handling also differs; in SOA all services share the same data storage unlike in microservices architecture where every service can have its own data storage. The fundamental difference though is the size and the scope of the services. Microservices are significantly smaller than SOA´s services. (Miri 2016) Microservice often handles a single process whereas SOA´s service handles one or multiple functions.

Microservices architecture is the opposite of a monolithic architecture where a monolithic application is built as a single unit. Applications usually consist of three main parts: client-side user interface, database and the server-side application. "*This server-side application is a monolith – a single logical executable.*" And when making changes to the system a new version of it must be build and deployed. (Fowler & Lewis 2014) And when it comes to a large scale system, with a monolithic architecture there comes problems like incomprehensible code, increased deployment time, scalability issues and a long-term commitment to a technology stack. With microservices architecture these

issues don't exist. Small services are easy to understand, can be deployed and scaled independently and can be deployed on different technology stacks. Each microservice is its own unit, running in its own process independently. Different microservices of the same application communicate through lightweight mechanism like RESTFull APIs. (Balalaie, Heydarnoori & Jamshidi 2015)

2.3 Containers

Containers are a method of virtualization which has become widely popular since the emergence of Docker. In container-based virtualization the virtualization layer runs as an application within the operating system (Rouse 2014). When comparing containers with virtual machines, containers are much more lightweight. When it comes to virtual machines each virtual machine includes its own operating system as well as the application. With containers each container shares the host´s operating system kernel with other containers. (Rubens 2015) In the below picture the difference between virtual machines and Docker containers can be seen.



Picture 3. Virtual machines and Docker containers compared. (Docker 2016a)

Containers are self-sufficient immutable bundles which are run in an isolated process. They only contain components that the application needs. Usually these components are:

- Runtime libraries
- Application server
- Database
- Artifact (JAR, WAR, static files etc.)

(Farcic 2016, 45)

2.3.1 History of Containers

Docker has been very successful on popularizing containers and sometimes containers and Docker are used as synonyms. However there are other container technologies as well. Containers are not so new technology as you might think. The first steps towards container technology were taken in the early 2000s (Hildred 2000).

In the year 2000 early implementation of container technology *jails* was added to FreeBSD, the free Unix-like operating system. In 2001 Linux started working on container technology and the Linux-VServer was released. It was the first effort on Linux´s part to "*separate the user-space environment into distinct units (Virtual Private Servers) in such a way that each VPS looks and feels like a real server to the processes contained within."* Linux-VServers weakness was that it required patched kernel requiring more overhead on distributors and system administrators. (Hildred 2015)

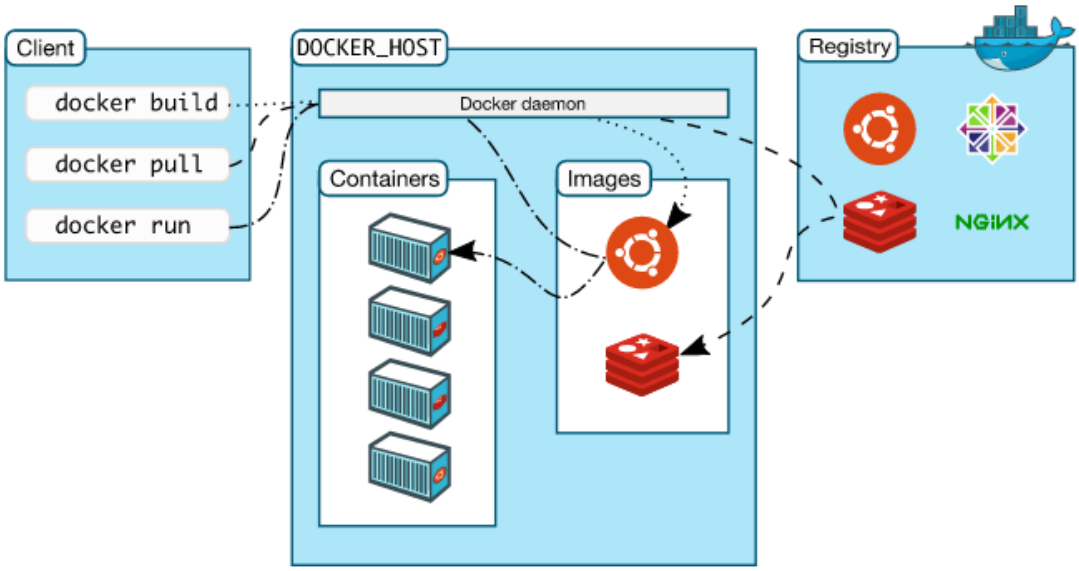
Innovations such as cgroups and kernel namespaces pushed container technology forward. Cgroups was invented in 2006 and it allowed processes to be grouped together ensuring that each group gets its own share of memory, CPU and disk I/O. Kernel namespaces and user namespaces particularly, which were invented in 2008, were an important invention in containerizations history. User namespaces allowed processes to have their own set of users and enabled a process root privileges inside a container. (Hildred 2015)

In 2008 came the LXC, project of Linux that layered user space tooling on top of cgroups and namespaces. The first release of LXC had some security issues but with the second release on 2014, the security improved. The open source Docker project built on top of all the innovations that came before it. The first version of Docker containers was released in 2013 (Avram 2013). Docker wrapped the LXC user space tools with easy-to-use tools "*aimed at developers looking for simple ways to package their applications*" (Hildred 2015).

2.3.2 Docker Containers

Docker uses client-server architecture. The Docker client talks to the Docker server called Docker daemon, which builds, runs and distributes Docker containers. The Docker client and server can run on the same host or the Docker client can connect to a remote

Docker daemon. The client and daemon communicate via sockets or through a RESTFull API. (Docker 2016b)



Picture 4. Docker architecture (Docker 2016b)

In the above picture the main components of Docker can be seen. In addition to Docker client and daemon there are three important resources:

- **Docker images:** Images are used to create Docker containers. Image is a read-only template that can for example contain an Ubuntu operating system with Apache and web application installed. Docker images are built with the `docker build` command.

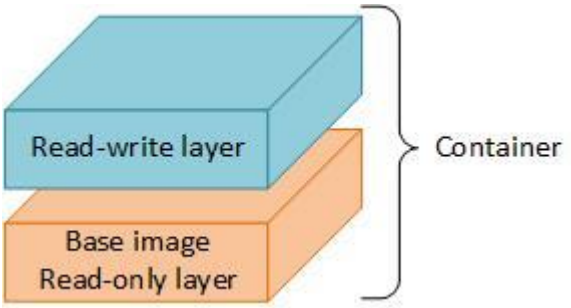
- **Docker registries:** Registries hold images. There are public registries like Docker Hub and private registries like Docker Registry and Docker Trusted Registry. The Trusted Registry is more secure enterprise-grade image storage solution. Docker Hub is large collection of existing images. To get images from Docker Hub or from the other registries the user uses the `docker pull` command. To store your own image to any of the registries the user uses the `docker push` command.

- **Docker containers:** Container holds everything that is needed for an application to run. Containers are created from a Docker image. To run a Docker container the user uses the `docker run` command. Other commands controlling

containers are for example `docker start`, `docker stop`, `docker create` and `docker rm` (to delete container).

(Docker 2016b)

The file system that Docker uses is called a Union File System. The file system is visualized in below picture.



Picture 5.Dockers Union File System.

Docker images are stored in a series of read-only layers. When container is started, Docker takes the read-only image and adds a read-write layer on top. If there are changes made in the container, for example an existing file modified, the file is copied out of the underlying read-only layer and into the top-most read-write layer where the changes are applied. The top-most read-write layer hides the underlying file, but does not destroy it; it still exists in the underlying image. Launching the same image will start a fresh container without any of the changes made in the previously running container. (Mouat 2014)

Other important concept with Docker is data volume, which is specially-designated directory within one or more containers (Docker, 2016b). With data volumes it´s possible to save or in other words persist data. Volumes are directories or single files that are outside of the default Union File System and exist as normal directories and files on the host file system. Data volumes are declared when container is run. This will make the directory or file inside the container live outside the Union File System and available on the host. (Mouat 2014)

3 BENEFITS AND DRAWBACKS OF MICROSERVICES ARCHITECTURE

To understand better what are the benefits from using microservices architecture the author interviewed David Kuridza who is a software developer in a Slovenian company called 3fs. 3fs is an IT company started in 2005 whose products include web and mobile applications. Probably the best known product of 3fs is Toshl which is a personal finance app for iOS, Android and Windows phones. 3fs is also working in the OneCloud development team and Kuridza has been developing the OneCloud API.

Kuridza has been part of the 3fs company since its beginning. He´s used microservices in few projects and has seen the benefits but also the drawbacks. One of the projects started with a monolithic architecture and when the project grew they began to slice it into smaller chunks. At first they modified it into service oriented architecture and when the system started getting performance impact then they sliced it into microservices. One part of the system generated a lot more traffic than the rest of the system so it made sense to split that part into its own service and scale up that service only. The project team was happy about their decision to switch to microservices architecture: *"In the end we started off by trying to solve the performance issues and we ended up with a lot of good stuff happening just because of this decision. "*

Other one of Kuridza´s projects started as a service oriented architecture with microservices in mind. The project team planned the system from the beginning in a way that it could be split into microservices later on if necessary. And that has also been the case in OneCloud development. This is a better practice than diving straight into microservices architecture. More on the subject in the 4th chapter *Best practices for developing with microservices and containers*.

3.1 Benefits

In Kuridza´s project the independent scaling has been a clear benefit with microservices architecture. Other benefits include independent life cycle and ease of maintenance. With independent life cycle developer can deploy or make a change to a single service without breaking the whole system. Though Kuridza notes that deploying independently

is not as easy as it sounds because many services use the same datasets. So developers have to keep that in mind when deploying new services.

Martin Fowler who is an author, international public speaker and a software developer at ThoughWorks, a software delivery and consulting company, also mentions the ease of maintenance as one of the benefits of microservices architecture. When the system is divided into microservices, a single developer who wants to make a change doesn't need to understand the whole system and maintaining the system becomes easier. (Fowler 2015)

Newman, the author of the book *Building Microservices* (2014), mentions technology heterogeneity as one of the key benefits from microservices. Microservices are independent and there for it´s possible to use different technologies inside each one. The development team can pick the right tool for each job. (Newman 2014, 20.) As good as this sounds, this may have a negative impact on team work as Kuridza mentions in his interview. More on the difficulty of technology heterogeneity from team work perspective later in this chapter.

3.2 Drawbacks

With benefits there's also drawbacks. A fundamental drawback Kuridza mentions, with microservices architecture, are networking problems. Usually microservices architecture is developed with APIs on top of every single service. Microservices are then communicating through those APIs. When the APIs are communicating with each other and making requests between two different nodes, the system can easily get more traffic than the capacity can handle. Other networking problem is that networks fail and to handle these failures developers need to build a lot of logic in between.

Kuridza also points out that microservices also need service discovery tools. This isn't necessarily a problem, it's just something that needs to be taken to a count. In order for different microservices to make requests to each other they need to know the network location of the service instance they are making the request to. In modern cloud-based microservices architecture service instances have dynamically assigned network locations. Also, because of auto-scaling, failures and new upgrades, the set of service instances changes dynamically from time to time. (Richardson 2015)

Service discovery tools manage services and make it possible for the services to find each other. Key part of service discovery is a service registry which is basically a database containing network locations of service instances. Service discovery tools provide health checks and load balancing for the services. (Richardson 2015) More on service discovery tools is discussed on the next chapter.

Other technical difficulty Kuridza mentions is the problem with using same datasets, as already mentioned, which can be handled with better communication across the teams. And finally the difficulty of monitoring, logging and metering of the system since it is a distributed and complex system. There also needs to be a complete CI/CD pipeline in place in order to make reliable integration tests.

From teamwork perspective there are also drawbacks. Microservices allow every single service being written in different language and using different technologies. From competence perspective this can be a problem. Newman acknowledges this difficulty as well (Newman 2014, 20). If each microservice is very unique from development aspect and one person is developing a certain microservice then that person becomes irreplaceable. If he or she leaves the development team, finding another developer with the same competence can be difficult. At 3fs they have solved this issue by allowing only few technologies.

Other issue from teamwork perspective Kuridza remarks is that communication plays a larger role when developing microservices. Each microservice has an independent lifecycle but whenever a developer is making changes to a service she or he needs to make sure that everybody on the development team understands those changes and knows what effects it has to other services.

3.3 Summary

To summarize; benefits from microservices architecture are independent lifecycle and scaling, ease of maintenance and making new releases and finally no dependence on one technology stack. The drawbacks are networking issues and other issues that bring a lot of overhead, such as datasets, service discovery and teamwork. In Kuridzas opinion the benefits outweighs the drawbacks. There's a workaround for every drawback, it just brings overhead for the development team. Microservices are not the best solution for

every system and there are many aspects to be considered before starting to develop with microservices.

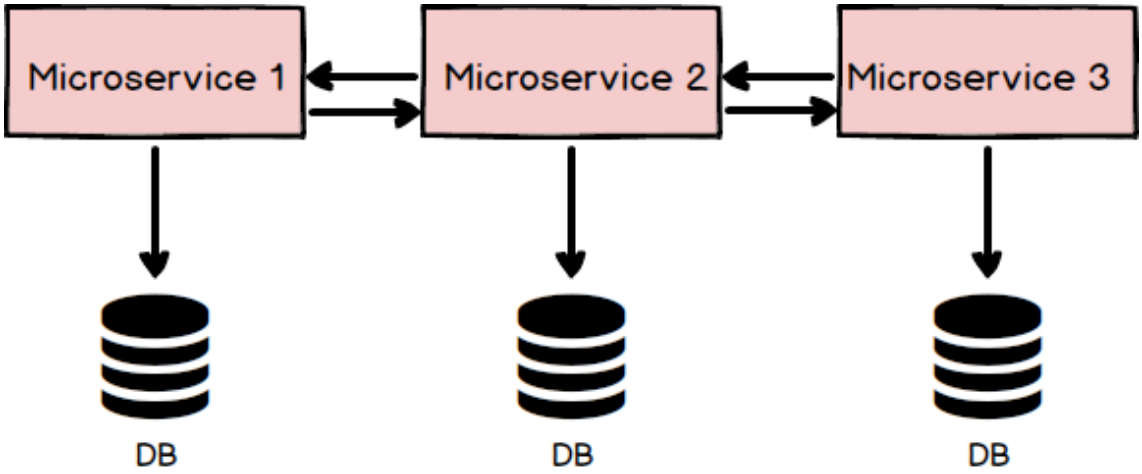
4 BEST PRACTICES FOR DEVELOPING WITH MICROSERVICES AND CONTAINERS

Viktor Farcic, a Senior Consultant at CloudBees[1] and the author of the book *The DevOps Toolkit 2.0 – Automating the Continuous Deployment Pipeline with Containerized Microservices* (2016), gave a presentation to Ericsson´s OneCloud projects team on 23rd of August 2016. In his presentation he touched upon the subject of best practices in microservices architecture. He stated that: *"There are no good practices. What is a good practice today isn´t that tomorrow. The technology is moving so fast."* New tools to facilitate developing with microservices appear every so often and supersede the old tools. However there is still some fundamental practices that are good to keep in mind when developing with microservices.

In his interview Kuridza says that he favors the practice of starting the development of a new system as a service-oriented architecture. This is justified firstly because at the beginning of a new development project it is still unknown how the system is going to grow. For example which parts of the system needs scaling up or scaling out. When the system starts to get performance impact, it´s easier to see which parts of the system need to be split into microservices. Secondly in order to develop with microservices architecture there needs to be a great backbone in place. That backbone includes monitoring, logging and metering tools, microservice creation and deploying tools, defined protocols and service discoverability.

A second good practice which both Farcic and Kuridza mentions is that each service or microservice has its own database. If one microservice needs another database than its own it has to go through another microservice. The setup is visualized in below picture.

[1] https://www.cloudbees.com/



Picture 6. Microservices each with its own database.

By following this practice the development team can make sure that the data in every database stays valid. If more than one microservice would share the same database and make changes to the data at the same time, it could invalidate the whole database and the data could no longer be trusted.

Farcic is a strong supporter of test-driven development (TDD). In his presentation he stated: *"Do TDD or die!"* In his opinion a product doesn´t have a change to succeed if TDD isn´t used during the development. In TDD process the tests are written first and the implementation code after that. The procedure goes like this:

1. Write a test
2. Run all tests
3. Write the implementation code
4. Run all tests
5. Refactor

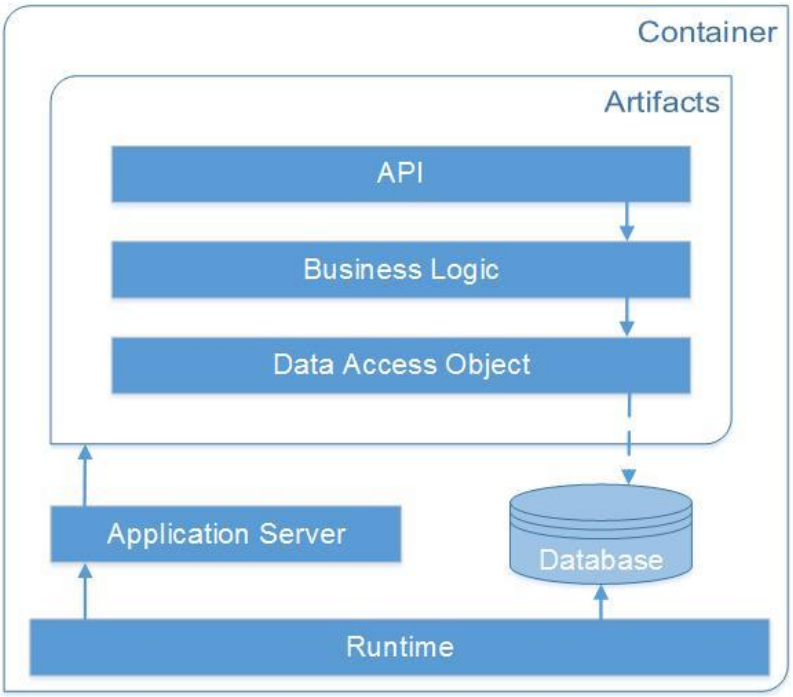
This procedure is usually called Red-Green-Refactor. Since tests are written before the actual implementation, the test are supposed to fail, hence the red state. In the green state the implementation code is written, but the amount of code that the tests pass. If the test don´t pass the implementation is wrong and should be corrected. In the refactor state improvements are made to the existing code without introducing any new features. When refactoring, all test should be passing, if not, refactoring has broken an existing functionality. (Farcic 2014)

Tests are not the only benefit from TDD. TDD allows developing without being afraid that something will break. With TDD in place the developer is able to reorganize the code while having the confidence that no functionality will be broken. This improves also the quality of the code. In fact: "*The main objective of TDD is code design with tests as a very useful side product.*" (Farcic, 2014)

A final good practice with microservices which Kuridza mentions is to enable microservices to do synchronizing work. Frontend can invoke tens of HTTP requests at the same time. Those requests get queued up and it takes time to receive response data. However for the user it has to look like the system is running smoothly and his request was processed. By enabling microservices to do synchronizing work, the microservice sends a response to the frontend even though it hasn´t processed the request yet. To the user this will look like his or her request was processed straight away. Farcic suggests the use of proxy microservices to handle this same problem. Proxy microservices invoke different microservices and return an aggregated service. They group several responses together and respond with aggregated data to the user. (Farcic 2016, 47)

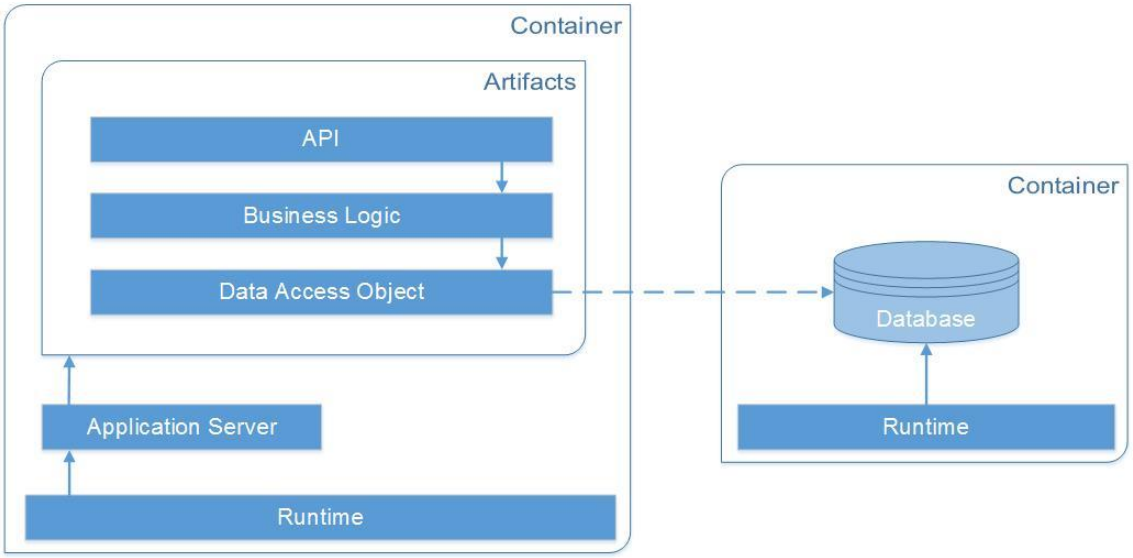
Farcic says that microservices should be as minimal as possible, containing only packages, libraries and frameworks that they truly need. This is true with containers as well. When choosing the OS for the container CoreOS might be a better solution than a heavier OS like Ubuntu or RedHat. (Farcic 2016, 48) CoreOS is a bare boned operating system based on the Linux kernel which provides only the minimal functionality needed for deploying applications inside containers. (CoreOS, 2016)

When it comes to containers and their databases Farcic suggests to use external databases for containers rather than have the database inside the container. Below both setups are explained.



Picture 7. Self-sufficient microservice inside a container (Farcic 2016)

In the above setting the container is self-sufficient with the database embedded into the container. This setting poses a problem when it comes to scaling. When scaling this kind of container on multiple nodes in a cluster, the databases should be either synchronized or their data volumes should be located on a shared drive. Data volumes are explained more thoroughly on chapter 2.3.2. When synchronizing the databases the container becomes unnecessarily complex and shared data volumes can have a negative impact on performance. (Farcic 2016) The alternative is presented in the below picture.



Picture 8. Microservice inside container with the separate database (Farcic 2016)

In the above setting the database has been externalized into a separate container. This demands that each service has two different containers linked together preferably through a proxy service. This setting increases slightly deployment complexity but nevertheless provides greater freedom when scaling. (Farcic 2016)

Following these best practices is a good starting point when beginning the development with microservices and containers. However there are still many other things to consider. For example how to do service discovery or monitoring of microservices. In this thesis the author won´t elaborate on those subjects since it would involve comparison of tools which can be outdated quite fast.

5 CASE: ERICSSON ONECLOUD

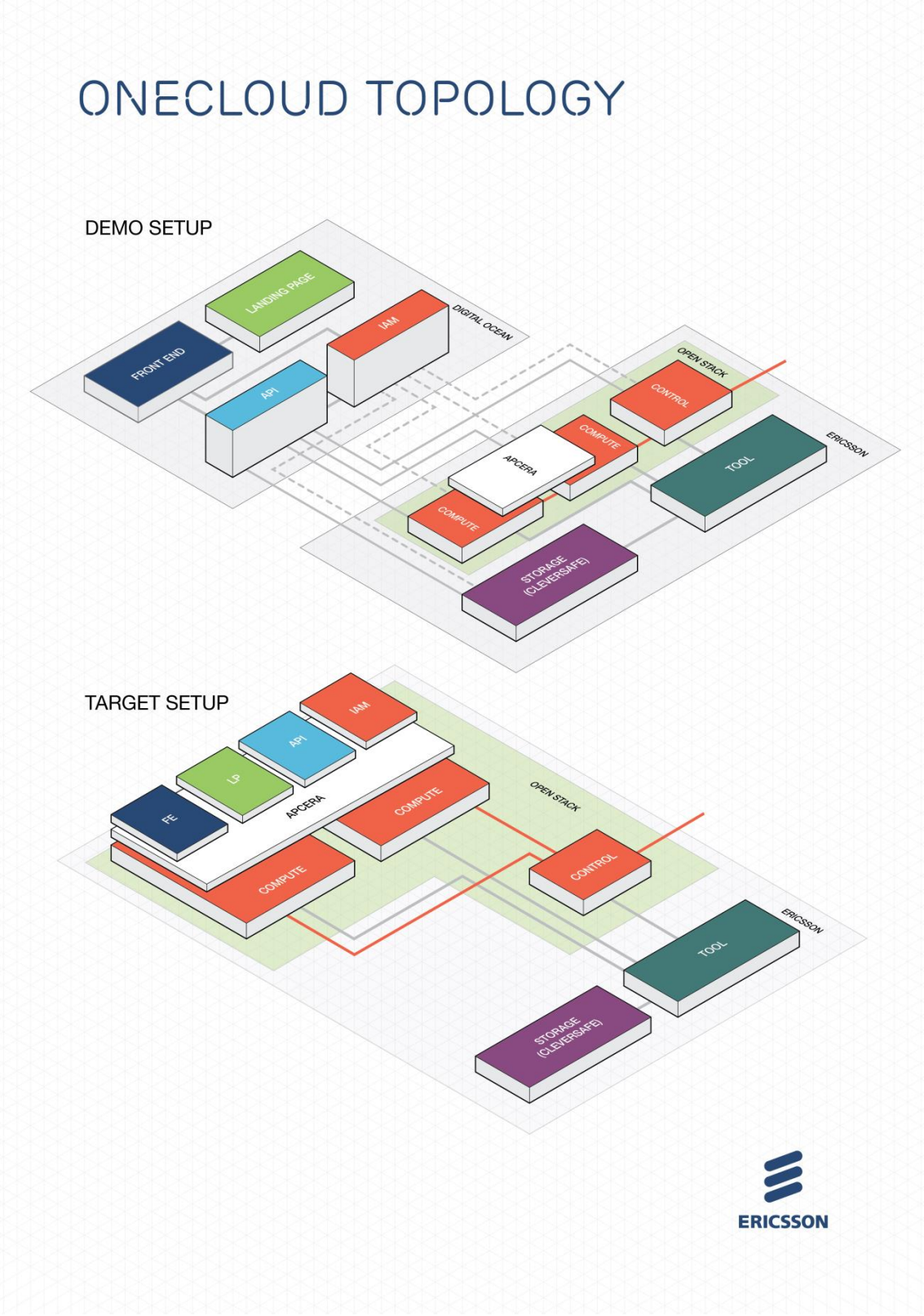
Microservices and containers will take an important role in Ericsson´s OneCloud project. Projects aim is to build a public hybrid cloud. The development team consist of approximately 30 developers working in Finland and in Slovenia. The complexity of the development and the physical distance between developers brings quite many challenges to the development. Microservices and containers can facilitate few of those challenges. The development has been divided in five teams:

- Frontend
- Application Programming Interface (API)
- Identity & Access Management (IAM)
- Infra
- Continuous Integration & Continuous Deployment (CI/CD)

This chapter concentrates more on the OneCloud API but first the whole setup of the of the OneCloud environment is explained.

5.1 The OneCloud Environment Setup

In below picture is the demo setup which was implemented for the projects 1st phases' demo. In the picture is also the target setup which will be run on top of Ericsson's Hyperscale Datacenter System 8000 (HDS-8000).



Picture 9. OneCloud's setup for the demo and for the finished product.

In the demo setup there was a tool node, object storage solution Cleversafe, Openstack and Apcera running on four HP servers and two Dell servers while frontend, landing page, API and IAM´s API was running in cloud infrastructure Digital Ocean. All of the components are deployed on virtual machines or containers.

The tool node, which can be seen from the picture 9 above, is used for accessing the other components or VMs of the OneCloud with SSH connection. The tool node is in fact a combination of tools such as Bastian Host, Fuel and Terraform.

Cleversafe is the object storage solution of OneCloud which manages data as objects. It can manage massive amounts of unstructured data. After the demo MySQL and MongoDB databases were also deployed in containers to the storage node. The goal is to have a block storage solution Nexenta deployed which will also enable the use of MySQL and MongoDB.

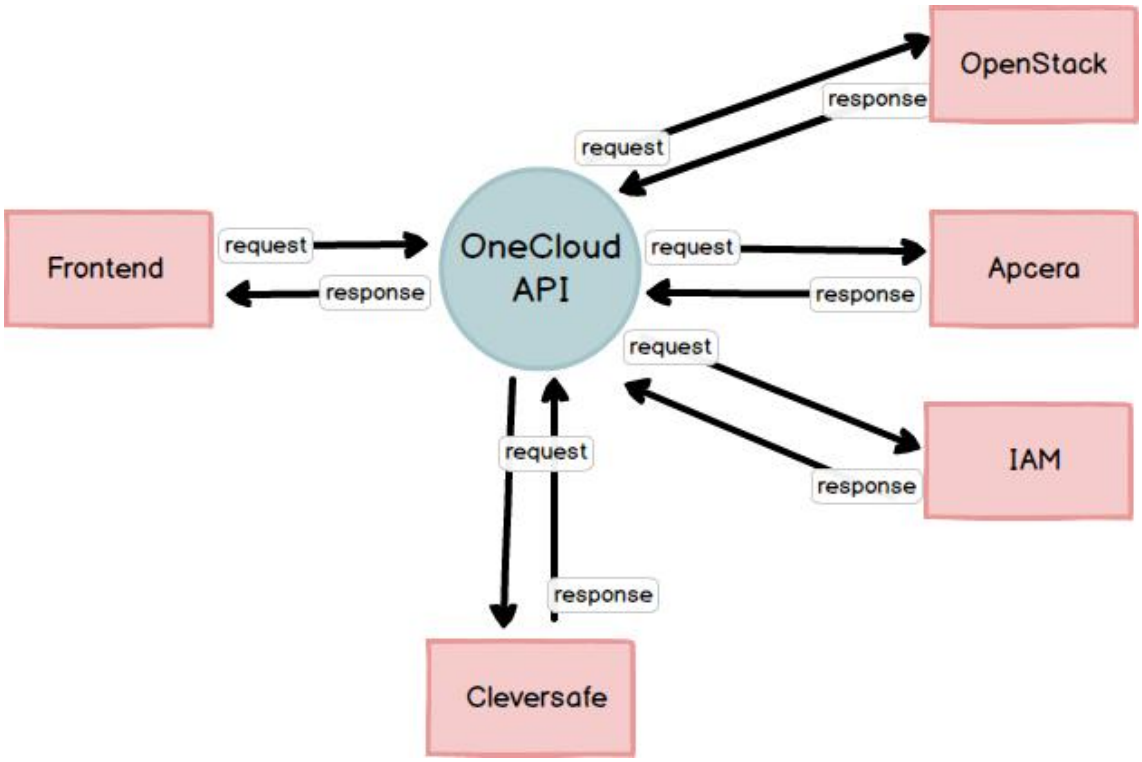
OpenStack is a free open-source software platform for cloud computing. In OneCloud OpenStack is deployed as an infrastructure-as-a-service which runs VMs. Apcera is a platform-as-a-service cloud computing platform needed for running containers. It is also used for policy control for the containers.

The goal is that in the target setup all the components would be running on Ericsson`s Hyperscale Datacenter System 8000 (HDS-8000). In distributed computing environments, such as cloud, hyperscale computing is required in order to scale the system efficiently. Hyperscale computing provides the ability to smoothly provision and add compute, memory, networking and storage resources to a given node or set of nodes. (Techopedia, 2016)

HDS-8000 is Ericsson's solution for hyperscale computing. "*HDS-8000 provides the compute power, storage capacity, and networking capabilities to host hyperscale cloud platforms, plus the control and analytics software to monitor and manage them.*" (Ericsson, 2016b) HDS-8000 uses Intel's rack scale architecture which makes it possible to split up the compute, storage and network resources making it more flexible and resource efficient (Ericsson, 2015). HDS-8000's optical backplane provides high-speed connection between the compute, storage and networking resources. Compared to electrical backplane in HDS-8000 distance and capacity limitations are non-existing. (Ericsson, 2016c)

5.2 OneCloud API

OneCloud API serves as an interface between the different components of the OneCloud; frontend, IAM's API, Apcera, Openstack and Cleversafe, and it facilitates their interaction. The API is a service which will be later on split into several microservices. It consists of multiple endpoints which are unique URLs that represent an object or collection of objects. The interaction happens with Hypertext Transfer Protocol (HTTP) request messages. Below picture visualizes the interactions between different components.



Picture 10. OneCloud API and the interaction between different components.

Here's an example of a HTTP GET-request that retrieves a list of virtual machine locations.

```
GET https://api.eclouddemo.io//v1/compute/vms/locations

Connection: keep-alive

Authorization:                                                    Bearer
eyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxm0
```

```
Host: api.eclouddemo.io
```

The response data from the above request comes in JavaScript Object Notation (JSON) format:

```
{
  "locations": [
    {
      "id": "dd3bd64b-b9fe-49fb-b509-e7342f62f6b4",
      "name": "Jorvas",
      "available": true,
      "_links": {
        "self":
"https://api.eclouddemo.io/v1/compute/vms/locations/dd3bd64b-b9fe-
49fb-b509-e7342f62f6b4"
      }
    }
  ]
}
```

The OneCloud API is a representational state transfer API (REST API). REST APIs have these main characteristics:

- **Client-server:** The client is in charge of the frontend and the server of the backend. Both are independent from each other.
- **Stateless:** Client data is not stored on the server between requests and session state is stored on the client.
- **Cacheable:** Clients can cache response to improve performance.

(Deering, 2012)

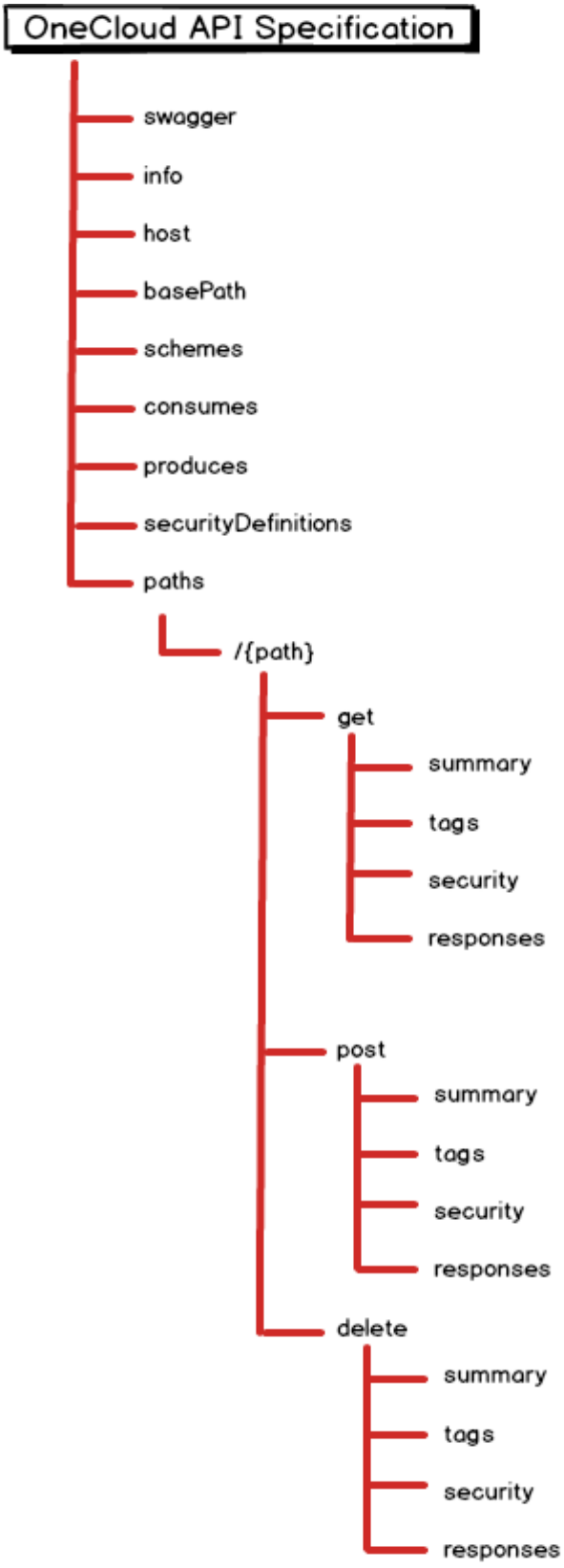
The OneCloud API development started from producing a specification which is following the OpenAPI Specification guidelines. The OpenAPI Specification is overseen by the OpenAPI Initiative, an open-source collaborative project of the Linux Foundation. The goal of the project is *"to define a standard, language-agnostic interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic*

inspection." (OpenAPI Initiative, 2016) The OpenAPI Specification is an API description format or API definition language (Lauret, 2016).

In OneCloud APIs specification we describe the following things:

- Basic information about the API
- Available paths i.e. endpoints i.e. resources
- Available operations on each path; meaning the possible HTTP request verbs that can be used i.e. GET, POST, PATCH and DELETE)
- Input/Output for each operation; required properties of the request, response code, required properties of the response and an example response

OneCloud APIs specification is written in YAML format which is a human-readable data serialization language. In the below picture the structure of the specification has been visualized. The specification goes even more into detail but here is the base structure.



Picture 11. The OneCloud API specification.

Ones the specification was ready the development of the API could begin. The development was done using Go as the programming language. Go is an open-source programming language created at Google (Pike, 2012). The development of the API proceeded hand in hand with the infra team. Whenever the infra team deployed new components to the system, the API team updated the specification and programmed new endpoints to the API. The specification facilitated developers work and when there was a specification reviewed by the whole team, everybody was on the same page on what was needed from the endpoint.

The API is build using a Makefile. Makefile is a file which contains shell commands and is named a Makefile. Make is a Unix utility that executes the Makefile. In a terminal, while in the directory containing the Makefile, the user can type `make help` and make will print out all the possible execution options. Here's the OneCloud APIs execution options:

```
api/build                       Builds API
api/run                         Run API
bootstrap                       Boostraps development environment
build/docker/builder            Builds docker image containing
                                dependency for building the project
build/docker/docs               Builds docker image containing
                                dependency for building the API
                                documentation
builder/%                        Runs make target in builder container
                                (example: make builder/console)
docs                            Builds API documentation
docs/build                      Generates the files for API
                                documentation
mock/build                      Build mocks
mock/clean                      Remove mocks
qa                              Performs QA checks
setup                           Sets up development environment
spec/build                      Builds API specification file
spec/test                       Validates API specification file
test                            Run tests
```

With the Makefile we can build a container with the API and all its dependencies. The Makefile has also some commands that run tests and commands for building API

documentation. There are three commands needed to make the API run; `make setup`, `make api/build` and `make api/run`.

The container running the API is build using a Dockerfile which is included in the code repository. Dockerfile is a text document containing commands that are needed for building a container. Those are the commands that the user would type in a terminal but with the Dockerfile those are run automatically in a series. Here's the Dockerfile that is needed for building the API:

```
FROM golang:1.6

RUN \
    apt-get update && \
    apt-get install -y \
        npm jq

RUN npm install --global json-refs
# fix apt's naming error, see following URL for details:
#  https://github.com/nodejs/node-v0.x-archive/issues/3911#issuecomment
-8956154
RUN ln -s /usr/bin/nodejs /usr/bin/node

# Install go dependencies
RUN \
    go get -u \
        github.com/kardianos/govendor \
        github.com/golang/lint/golint \
        github.com/go-swagger/go-swagger/cmd/swagger \
        github.com/smartystreets/goconvey \
        github.com/golang/mock/gomock \
        github.com/golang/mock/mockgen

ENV TERM xterm-256color
RUN                                echo                                'export
PS1="${debian_chroot:+($debian_chroot)}\[\033[01;32m\]\u@\h\[\033[00m\
]:\[\033[01;34m\]\w\[\033[00m\]\$ "' >> ~/.bashrc

# Cleanup
RUN apt-get clean
RUN rm -rf /tmp/* /var/tmp/*
RUN rm -rf /var/lib/apt/lists/*
```

With the Makefile and Dockerfile the API deployment can be made very efficiently. The person deploying the API doesn't need to understand the code, the Makefile or the Dockerfile. He or she just needs to run the appropriate commands. The container running the API has all the dependencies and is an independent unit, so there's no worry about the environment where the container is running in. This is the beauty with microservices

and containers; easy deployment, independent processes and no need to restrain to one technology stack.

5.2.1 The Future Development of the OneCloud API

In current development there is no test-driven development in place. If we´re to believe Farcic, TDD has to be introduced as a way of working. The authors view is that TDD hasn´t been used in the development because it is seen as an unnecessary task with too much overhead. However it would seem that TDD doesn´t slow down the development, it could help the development team to improve the quality of the code and reduce the time spent on refactoring.

As already mentioned at the moment the OneCloud API is a service rather than a microservice. It serves one purpose; handling the interaction between different components, while having several functionality. Later on it will be split up to several microservices. Every endpoint or even every endpoints method (HTTP request method GET, POST, PATCH or DELETE) could be a single microservice. How it will be split up, is determined by the complexity of an endpoint. If an endpoint is more complex, for example virtual machine endpoint which handles retrieving, creation and deletion of virtual machines, it's better to split it. Locations endpoint for example, which handles only the retrieving of resource locations, isn't that complex and it won't change much during time. Therefor the locations endpoint can be one microservice.

But before the API can be split into microservices, there needs to be great many tools in place; the backbone of the microservices architecture. Service discovery and monitoring tools being the most important once.

6 CONCLUSION

The goal of this thesis was to investigate the true benefits of microservices architecture and whether or not the benefits outweigh the drawbacks. The goal has also been to study the best practices for developing with microservices and containers.

When it comes to benefits what I´ve found is that the key benefits from microservices architecture are; ease of maintenance, independent scaling, independent life cycle, and technology heterogeneity. Some of the benefits comes with drawbacks and aren´t as straight forward as it would seem. For example with technology heterogeneity, the possibility to choose from large amount of tools and technologies can be troublesome. The development team has to make sure that there is competence to build the microservices with chosen technologies now and in the future.

Other drawbacks from microservices architecture I´ve found are technological hindrances or team work and competency related. There are lot of new technological issues that aren´t a problem in a monolithic system. For example growing demand on networking capacity. These issues demand a new type of methods and solutions for the development. There are lot of decisions to be made before starting the development with microservices.

The author has found during her studies that the technology around them is constantly developed and improving. Therefor it´s no use to recommend tools to use for developing with microservices and containers. However good practices that I´ve found are; to start the development from service-oriented architecture and then build up to microservices architecture, to build as minimal microservices and containers as possible or needed, to have an own database for each microservice, to use test-driven development and finally to use proxy microservices to handle the large number of HTTP requests.

Finally it can be stated that with this thesis I´ve only scratch the surface of microservices architecture and it would demand more studies and exploration to see the benefits from microservices in practice and to learn more on the best practices. However this thesis can be used as a starting point when beginning the development with microservices and containers.

REFERENCES

Avram, A., *Docker: Automated and Consistent Software Deployments*, March 27, 2013, <https://www.infoq.com/news/2013/03/Docker>.

Balalaie, A., Heydarnoori, A. & Jamshidi, P., *Migrating to Cloud-Native Architectures Using Microservices: An Experience Report*, 2015.

CoreOS. 2016. *Documentation.* Retrieved September 4, 2016. <https://coreos.com/docs/>.

Deering, S., *Do you know what a REST API is?,* December 7, 2012, <https://www.sitepoint.com/developers-rest-api/>.

Docker. 2016a. *What is Docker?,* Retrieved September 2, 2016. <https://www.docker.com/what-docker>.

Docker. 2016b. *Understand the Architecture.* Retrieved July 6, 2016. <https://docs.docker.com/v1.11/engine/understanding-docker/>.

Ericsson, *Company Facts*. Retrieved July 1, 2016. <https://www.ericsson.com/thecompany/company_facts>.

Ericsson, *Ericsson introduces a hyperscale cloud solution.* 2015. <https://www.ericsson.com/res/docs/2015/intel-ericsson-solution-brief.pdf>.

Ericsson, *Introducing Ericsson Hyperscale Datacenter System 8000,* Retrieved August 9, 2016. <http://www.ericsson.com/hyperscale/cloud-infrastructure/hyperscale-datacenter-system>.

Ericsson, *Optical backplane.* Retrieved August 9, 2016. <http://www.ericsson.com/hyperscale/cloud-infrastructure/hyperscale-datacenter-system/optical-backplane>.

Farcic, V. September 30, 2014. *Test-Driven Development (TDD).* < https://technologyconversations.com/2014/09/30/test-driven-development-tdd/>.

Farcic, V. 2016. *The DevOps 2.0 Toolkit – Automating the Continuous Deployment Pipeline with Containerized Microservices.* Leanpub.

Fowler, M. & Lewis, J., *Microservices – a Definition of this new Architectural Term*, March 25, 2014, <http://martinfowler.com/articles/microservices.html>.

Fowler, M., *Microservices Resource Guide*, Retrieved June 30, 2016. <http://martinfowler.com/microservices/#who>.

Fowler, M., *Microservice Trade-offs,* July 1, 2015, <http://martinfowler.com/articles/microservice-trade-offs.html#deployment>.

Fowler, M., *ServiceOrientedAmbiguity,* July 1, 2005, <http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html>.

Hildred, T., *The History of Containers*, August 28, 2015. <http://rhelblog.redhat.com/2015/08/28/the-history-of-containers/>.

Kress, J.; Maier, B.; Normann, H.; Schmeidel, D.; Schmutz, G.; Trps, B; Utschig-Utschig, C. & Winterberg, T. July 2013. *Enterprise Service Bus.* <http://www.oracle.com/technetwork/articles/soa/ind-soa-esb-1967705.html>.

Lauret, A., March 2, 2016. *Writing OpenAPI (Swagger) Specification Tutorial – Part 1 – Introduction,* <https://apihandyman.io/writing-openapi-swagger-specification-tutorial-part-1-introduction/>.

Miri, I., July 5, 2016. *Microservices vs. SOA.* <https://dzone.com/articles/microservices-vs-soa-2>.

Mouat, A., December 9, 2014. *Understanding Volumes in Docker* , <http://container-solutions.com/understanding-volumes-docker/>.

Newman, S. 2015. *Building Microservices.* Sebastopol, USA: O´Reilly Media.

OpenAPI Initiative, *Specification,* Retreived August 12, 2016. <https://openapis.org/specification>.

Pike, R., 2012. *Go at Google: Language Design in the Service of Software Engineering,* <https://talks.golang.org/2012/splash.article>.

Red Hat, 2003. *Red Hat Enterprise Linux 3 – Reference Guide,* <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/3/html/Reference_Guide/ch-nfs.html>.

Richards, M. 2016. *Microservices vs. Service-Oriented Architecture,* 2016, Sebastopol, USA: O´Reilly Media.

Richardson, C., October 12, 2015. *Service Discovery in a Microservices Architecture,* <https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/>.

Rouse, M., 2014. *Container-based virtualization (operating-system-level virtualization)*, Retrieved June 28, 2016. <http://searchservervirtualization.techtarget.com/definition/container-based-virtualization-operating-system-level-virtualization>.

Rubens, P., May 20, 2015. *What are containers and why do you need them?*, <http://www.cio.com/article/2924995/enterprise-software/what-are-containers-and-why-do-you-need-them.html>.

Techopedia, *Hyperscale Computing,* Retrieved August 9, 2016. <https://www.techopedia.com/definition/28869/hyperscale-computing>.

Wilsenach, R., July 9, 2015. *DevOpsCulture,*
<http://martinfowler.com/bliki/DevOpsCulture.html>.