Aalto University
School of Science
Bachelor's Programme in Science and Technology

Security in Microservice Architecture

- Impact of a Switch from Monolith to Microservices

Bachelor's Thesis

xx. xxxxxkuuta 2020

Tommi Jäske

Aalto-yliopisto
Perustieteiden korkeakoulu
Teknistieteellinen kandidaattiohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

Tekijä:	Tommi Jäske
Työn nimi:	Turvallisuus mikropalveluarkkitehtuurissa - Monoliitisesta arkkitehtuurista siirtyminen mikropalveluarkkitehtuuriin ja sen vaikutukset.
Päiväys:	xx. xxxxxkuuta 2020
Sivumäärä:	?
Pääaine:	Computer Science
Koodi:	SCI3027
Vastuuopettaja:	Professori Eero Hyvönen
Työn ohjaaja(t):	Professori Tuomas Aura (Tietotekniikan laitos)
Kirjoitetaan myöhemmin.	
Avainsanat:	avain, sanoja, niitäkin, tähän, vielä, useampi, vaikkei, niitä, niin, montaa, oikeasti, tarvitse
Kieli:	Suomi

Aalto University
School of Science
Bachelor's Programme in Science and Technology

ABSTRACT OF
BACHELOR'S THESIS

Author:	Tommi Jäske
Title of thesis:	Security in Microservice Architecture - Impact of a Switch from Monolith to Microservices
Date:	MonthName 31, 2020
Pages:	?
Major:	Computer Science
Code:	SCI3027
Supervisor:	Professor Eero Hyvönen
Instructor:	Professor Tuomas Aura (Department of Computer Science)
Will be written.	
Keywords:	key, words, the same as in FIN/SWE
Language:	English

Contents

1 Introduction

In recent years, mobile applications and web services which cater to them have revolutionized our daily lives by infiltrating social life, shopping and almost every aspect of our existence. The rapid expansion and, at times, even faster decline of these web services needs a matching architecture to meet their very specific needs.

There are many web services already in use which were designed and implemented before the onslaught of microservices. Some of these services have already made the switch such as Netflix but this is not the case for the whole industry. Also, when a new service is being created the business domain might not be established yet. Additionally there might exist a fair amount of uncertainty in what exactly is to be developed. This is furthermore amplified by the use of agile software development methods in which the change in requirements is welcome even in later stages of the development (Beck et al., 2001).

When new development is carried out by a startup, the initial architecture might still be a monolith one. Newman (2019) states that, due to limited resources, a monolith might be a better fit to these companies trying to navigate to the actual product they are to offer. In the case of success, the need to rapidly scale the offering emerges. Newman (2019) refers to these companies as ”scale-ups”. Newman (2019) also states that it is much easier to refactor an existing service than to create a new one and thus the need to split monoliths to microservices is and probably will be relevant to the near future.

Kalske et al. (2018) finds that as the codebase becomes large the Monolith Architecture (MA) leads to slower development. This is due to the possible complexity inherint in the entwined monolith. The amount of code to refactor is much larger than in a small microservice. Microservice should do one thing and, as such, it should be more understandable.

The Stack Overflow annual survey (Stack Overflow) conducted on developers found that half of the respondents identified as full-stack or backend developers. 40% of the respondents had less than five years of professional experience.

New developers entering the workforce have a very different mindset than the older more seasoned professionals. Thus, it is very clear that the ways of working and paradigms to be used are constantly changing.

Microservices are not the proper choice for all web services (Newman, 2019). Microservices offer multiple benefits such as easier scalability and more modular structure for the application. When the architecture needs to be changed, the process needs to happen in an orderly and safe way. Often overlooked security aspects need to be identified and addressed as early as possible.

Microservice Architecture (MSA) differs in many ways from the more traditional Monolithic Architecture (MA). This shift entails very specific security issues.

In this thesis, the MSA and related security literature is surveyed and the main differences between MA and MSA on security aspects are discussed.

The first chapter discusses the ... The last chapter in the thesis contains the conclusions and presents further research topics.

2 Architectural Comparison

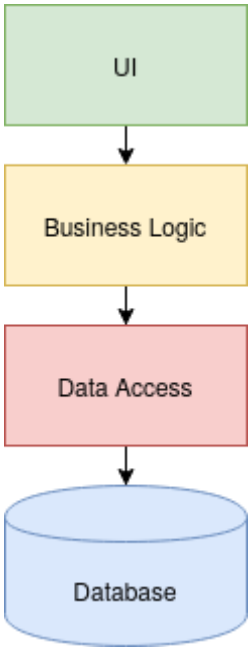

Figure 1: Traditional Monolithic Architecture (Kalske et al., 2018)

MA can be visually presented as in figure 1. The web service is a layered structure in which all of the different layers have a specific task to perform. This follows the Model-View-Controller (MVC) design pattern (Reenskaug, 2018). The UI is the View, the business logic is the Controller, and the database is the Model.

Fowler and Lewis (2014) define the microservice style as follows: the service is to be componentized using services in which a component is indepently replaceable and upgradeable, the services are organized around business capability rather a design pattern such as the previously mentioned MVC, a team should responsible of their product for it's full service life, the services are to contain the logic and communicate using a communication system without business logic "smart enpoints and dumb pipes", decentralized governance meaning that choices such as the technology to use or architecture are not dictated to developers, decentralized data management,

infrastructure automation, the whole application needs to be fault tolerant since individual services might fail or become unavailable, and evolutionary design.

Newman (2019) definition for a microservice is a service that: is independently deployable, is modeled around business domain, that owns the data that they need to operate, that communicates via network, is technology agnostic, that encapsulates data storage and retrieval and that has stable interface.

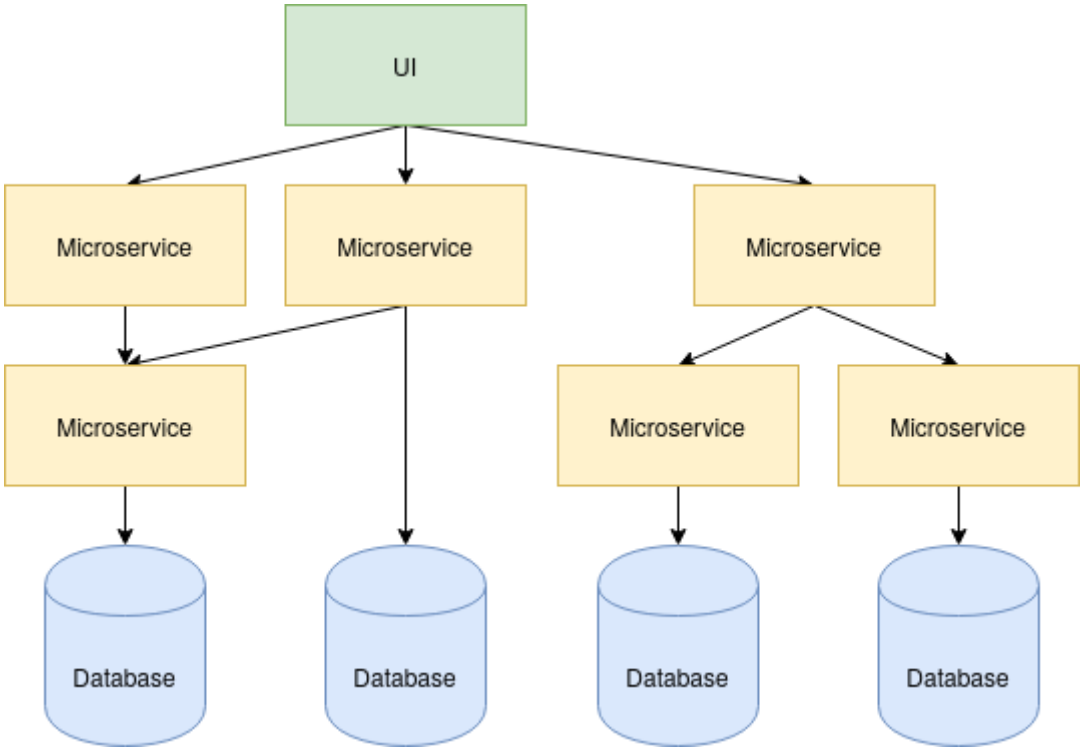

Figure 2: Microservice Architecture (Kalske et al., 2018)

An example for MSA, presented in figure 2, has many problem areas of which one is the challenging security implementation. This is due to the fact that every microservice accessible to the client can also be accessed or contacted by other more malicious parties in the same network. The network in the case of web services is the internet. The attack surface available for the malicious party is the entirety of the APIs offered by the microservices.

One solution to limit the attack surface is the addition of API Gateway to the architecture as in figure 3. Montesi and Weber (2016) present an API Gateway design pattern. In this pattern, there exists only one web service accessible to clients. The API Gateway allows for a natural place for a Policy Enforcement Point (PEP) and other more MSA specific features such as service discovery. The security features can be implemented in the API Gateway making it a critical component. Since all communication is to either flow through or be sanctioned by the API Gateway the performance and accessability are critical.

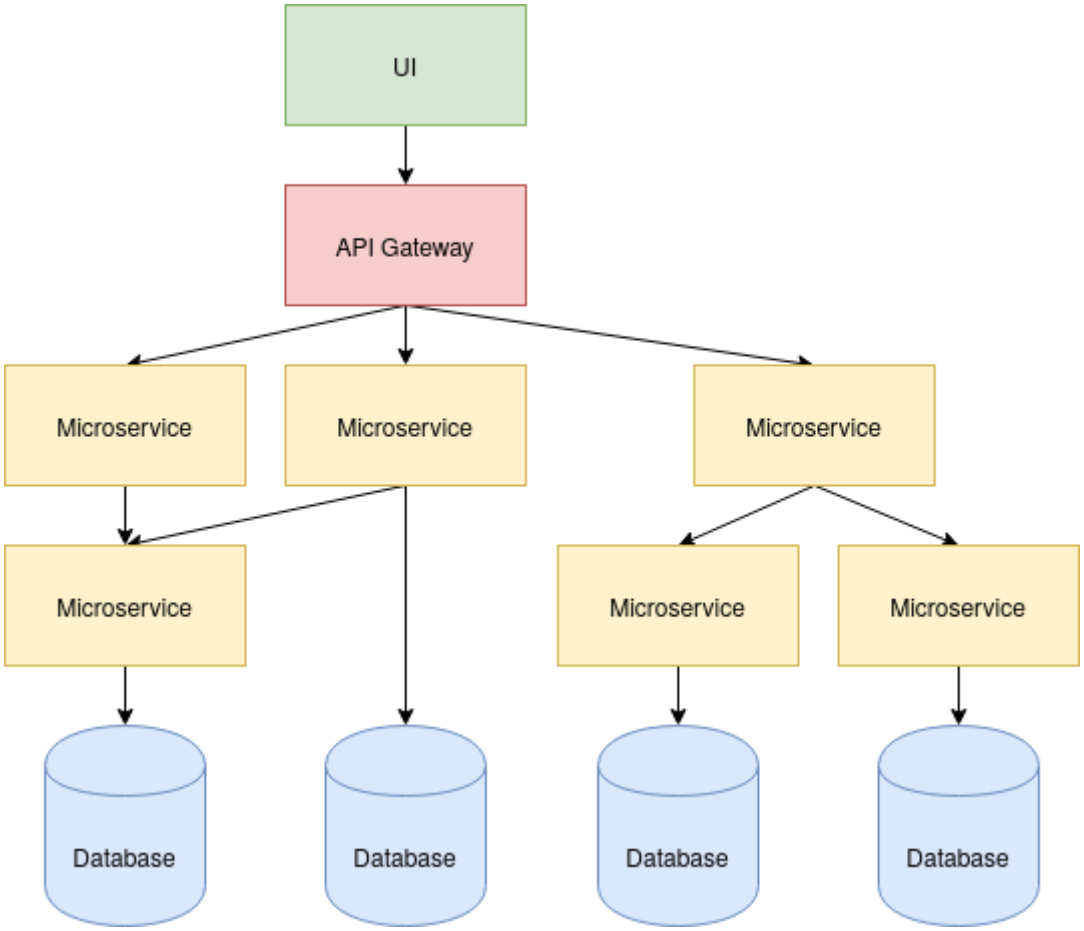

Figure 3: Microservice Architecture With API Gateway (oid, 2020)

3 Changing the architecture

The changing of the architecture of an already deployed service from MA to MSA should be a gradual process. This ensures a smooth transition and minimizes outages to the customers. Sometimes, though this is not possible. Newman (2019) states that when a monolithic application is implemented following a desing patterns such as the MVC (Reenskaug, 2018) this can lead to difficulties in the refactoring. The codebase is not split according to the business domain but follows a rigid design pattern.

In order for the process to be as simple as possible, the MA is or at least should be split into modules with separation of concerns (Yarygina, 2018). The actual splitting of the monolith can be carried out in various ways, one of which is Domain Driven Design (Evans, 2003). The selection of boundaries for the services is critical. If this is done incorrectly all the affected services need to be refactored to mend the error (Newman, 2019).

The MSA differs from a MA in fundamental ways. According to Fowler and Lewis (2014), one main difference is the communication between the components. In a

monolith application the processes can send function calls or method invocations amongst themselves. In MSA, the messaging is based on sending messages or HTTP requests. Function calls entail a stackframe creation in the call stack, execution of the function code, and finally popping the stackframe and returning the result. Compilers can optimize the code further and inline the function calls to eliminate the stackframe creation and the following procedures.

Communication using the network is extremely slow compared to local function calls. Zari et al. (2001) studied the response times of web sites offered to the public. The websites response times where measured in seconds. The requests sent to other microservices through the network are much slower than function calls within one computer. Therefore, the communication patterns should be changed to take into account the change in communication path. If the architecture is changed in such a way that the previous communication model amongst the components is preserved, there would be an excessive amount of communication and the resulting system would not be as performant (Fowler and Lewis, 2014).

4 Security

Richter et al. (2018) implemented a test system mimicking the Deutsche Bahn seat reservation system using MSA. The purpose of the test system was to analyze security risks that were introduced by the implementation. In the study they categorized the solution to three layers: first of which was the compute provider, the second was the encapsulation tehcnology, and the third one was the deployment. The technologies for these layers were: Amazon Web services, Docker for containers, and Kubernetes (k8s) nodes, respectively. They found out that the cloud-based infrastructure when used in MSA resulted in a more complex solution than in MA. The added layers such as the K8s, have to be configured correctly and an error in one could potentially compromise the whole system. In addiniton, implementation of security is very difficult and resource intensive. The rewards from a good security are invisible. When microservices are implemented or even planned the security should be taken into account as early as possible. Implementing security later in the project or as an after thought can be more expensive and very difficult.

Implementing security is hard. A theoretical proof for this can be found in Andersson (2001). His main finding was that an attacker has an advantage over the developers trying to defend a system. For the system to be secure the defending developers have to find all of the bugs where as the attacker has to find only one. Therefore, to make systems more secure the use of tools and frameworks that have been tested and are already in use is preferable to developing one's own solutions for a well known and already solved problem.

5 Access control

Access control consists of user authentication and authorization. A user has to authenticate him/her self and authorization is acquired to access to information or property.

In the cases where the user has to be authenticated, the web service needs a way to do this securily. There are many authentication schemes available but users prefer the password (Zimmermann and Gerber, 2020). Due to this the authentication is usually done using a tuple containing user credentials i.e. a username and a password. The user is authenticated and a key or token is transmitted to the user via the network. This communication in both MA and MSA, should be encrypted in a way that none of the actors in the transfer path can intercept the message and misuse the credentials.

The credential counterparts i.e. the secret shared by the server and the user have to be available for the web service for verification. When using MSA, the service should own it's own data. When ever information is available it is a target for thieves and hackers. The services in MSA are to be individually deployable and the service scalable. Authentication service implementation has to take this into account. The service has to adhere to practices that minimize the risks of data breaches.

As has been mentioned previously implementing security is hard and resource intensive. Therefore, the authentication implementation should adhere to an already existing framework or an another entity providing the authentication. This is the case for both the MA and MSA. The available choices include Lightweight Directory Access Protocol (LDAP), OpenID Connect (OIDC), Security Assertion Markup Language (SAML), and Kerberos.

5.1 Authentication and Authorization in MA

In MA, it is possible to implement features in such ways that a session can carry user information. This information can consist of the granted roles and rights for the user. This session can be queried when access control is needed to execute an action or operation or a specific authorization service or module can exist. An example of an web service implemented in MA is presented in figure 4. The user accessing the web service sends a request to load balancer which has all the information on the currently operational services. Each of the services is a single process in which an authentication and authentication service are present. When a user is first accessing the service credentials are to be verified and in a succesful case a session is created for the process.

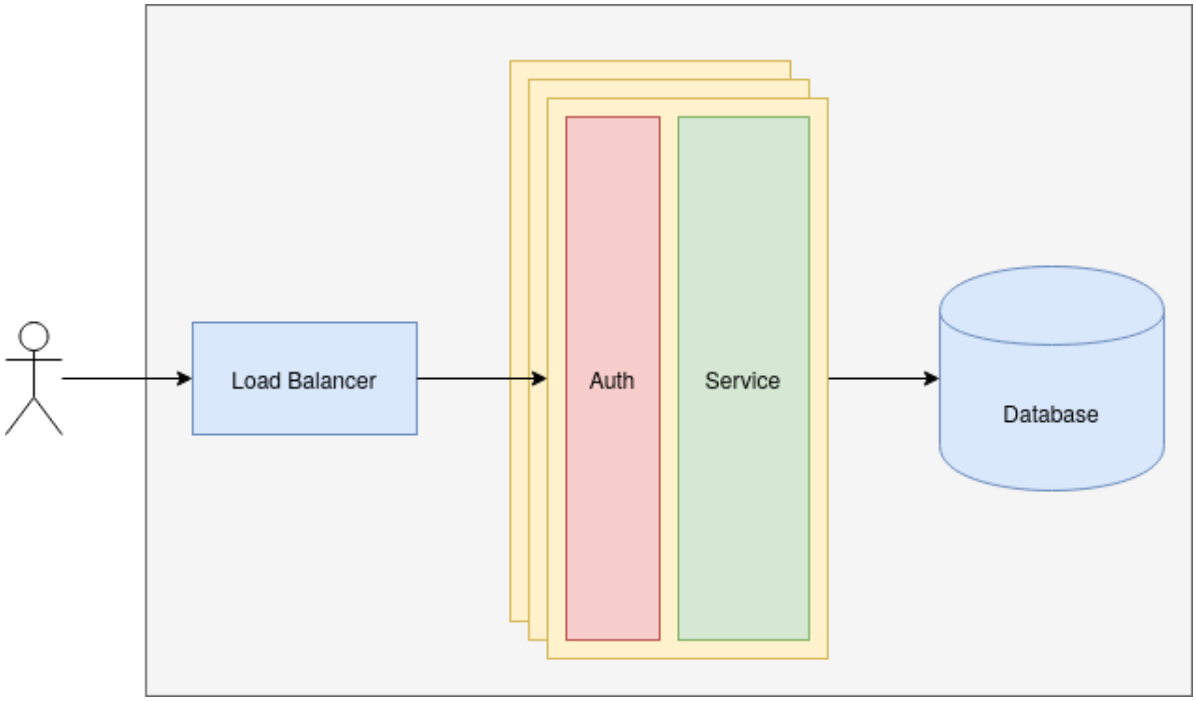

Figure 4: Traditional load balanced MA web service (He and Yang, 2017)

5.2 Authentication and Authorization in MSA

Authorization of the user rights can be implemented in various ways. One of which is an authorization service which can contain the access control matrix. Services being accessed verify from the authorization service that the client user or the role that the user has can access the requested service or functionality. In MA, the access rights to functionality can be implemented using annotations within the source code. The authorization is verified in memory and without any communication over the network.

In MSA accessing the access control matrix or matrices is not as easy as it is in MA. In order to verify that a specific right exists, the service would have communication with the authorization service. This communication would need to happen every time a user tries to access a functionality with access restrictions. This could potentially lead to extremely lively communication from all the services forming a bottleneck at to the authorization service. An example of such architecture is presented in figure 5.

5.3 OIDC

OIDC is an identity layer to accompany OAuth 2.0 authorization framework based protocol.

An example of an OIDC flow is presented in figure 6. The process flow steps are: the client application requests an authentication from the identity provider, the end user

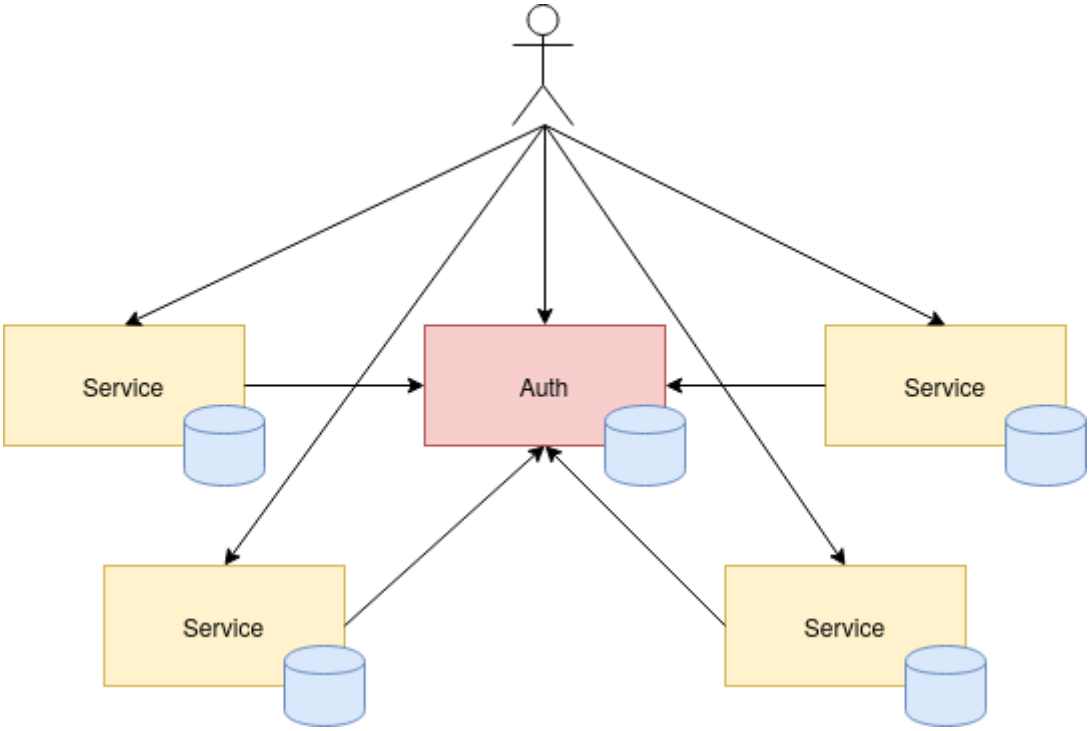

Figure 5: MSA authorization service (He and Yang, 2017)

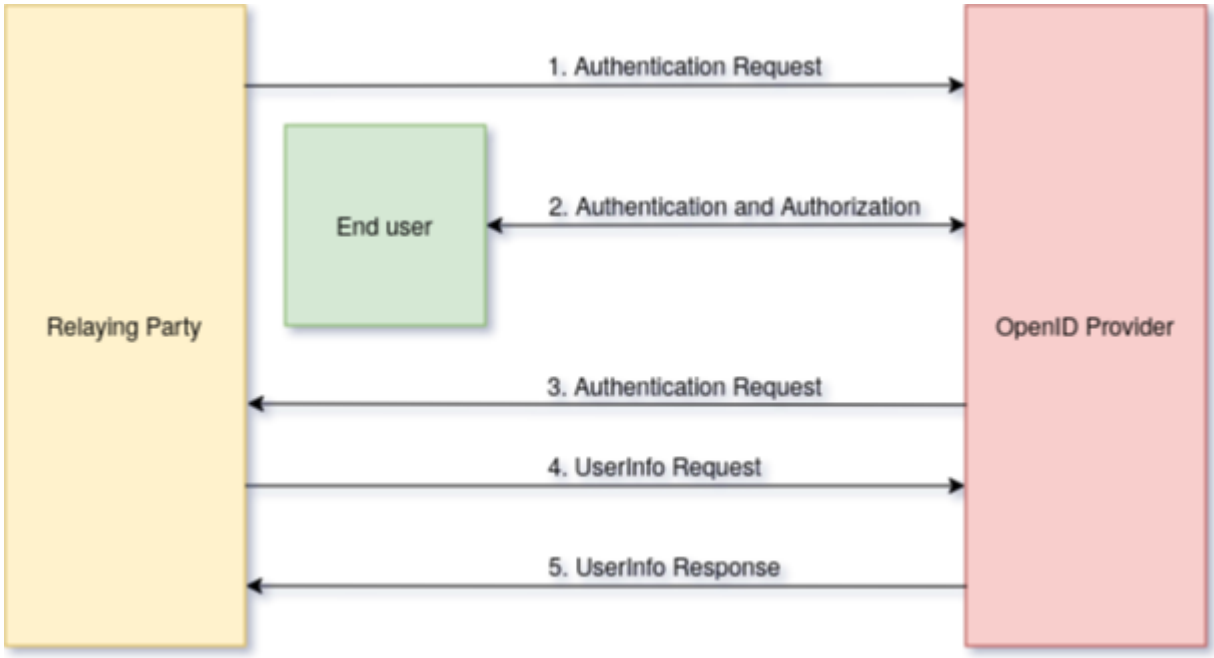

Figure 6: OIDC steps (oid, 2020)

is authenticated and authorization is obtained, the indetity provider responds to the initial authentication request by sending ID token and a possible access token, the client application can request the end user claims from the authorization server with the access token, and finally the authorization service returns the claims to the client application.

5.4 Java Script Object Notation Web Token (JWT)

JWT is a format to represent claims. It is base64 encoded, point separated strings, which can easily be carried in the HTTP request or response. The contents is key value pairs, and the token may or may not be signed and encrypted (Jones et al., 2015b). The token may contain an expiration time. If the token is used to validate requests without a server side implementation that can revoke a token, it will be valid until this time. The JWT token is issued by an authority trusted by the service. In figure 7 a basic bearer token based authentication and authorization is depicted. A client initially enters credentials to a login page and the authorization service issues a token for the user. This token is used in place of the credentials to access restricted resources.

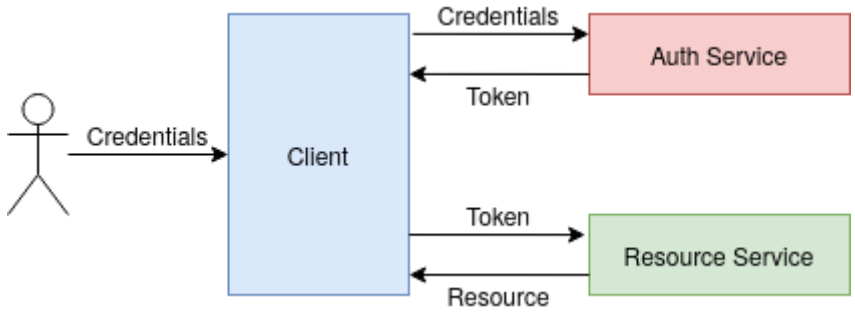

Figure 7: Bearer token based authentication (He and Yang, 2017)

The signing of JWT can be carried out in various ways. These are presented in the Jones et al. (2015a). The signature is computed using the algorithm and keys or certificates specified in the header values. When the token is signed using a private key it can be verified by all parties in possession of the public key.

5.4.1 Known Vulnerabilities

The choices for the algorithm for signing the JWT algorithm contain ”none” as one of the choices. This was found to be troublesome by McLean (2015). He found that many libraries did not operate in the desired way. The receiving party could be fooled to validate a mutated token without any signature with the ”none” as it's algorithm. In addition to this vulnerability McLean (2015) found that the verification suffered from another fatal flaw. When a token was created by using a symmetric algorithm, the servers could be fooled in to believing that a token signed by just the public key and not the secret HMAC key was a valid one.

5.5 Opaque Token

He and Yang (2017) compared several authentication and authorization solutions. One of the solutions discussed in the paper was the usage of an opaque access token for the client and API Gateway communication and map an Opaque token to JWT in the API Gateway. This would be used in all other communication as the means to autenticate and authorize the user. The main problem the opaque token is said to solve is the logout problem. A bearer token is valid until expiration and the client can not invalidate the token. The opaque token flow is presented in figure 8.

The use of two tokens has many advantages. The token granted to the client allows access only to the API Gateway. This token can not be used to access the services directly. Furthermore, the opaque token be can revoked by simply removing it from the storage for the token pairs. The usage of an internal JWT can alleviate some of the issues when MA is to be changed to MSA. Though not advisable, the JWT can carry session information and the whole session. This allows for more of the previous MA implementation to be used without as many changes.

6 Session

Client handled session is easy to tamper with and has risks involved. The service would either have to trust the client offered session or verify a signature. In both of these cases the client application would have to be aware of the service and would be able to communicate with it directly. This would bring considerable overhead on all the services and a security risk.

Sessions can be used in MSA when the architecture has an API Gateway. The session is stored in the Gateway and a session key (?) is carried in the requests made by the client application. The API Gateway then verifies the request and can grant access to specific service. Without the Gateway the session would have to be either centrally maintained at a session service or be handeld by the client and passed along in the requests.

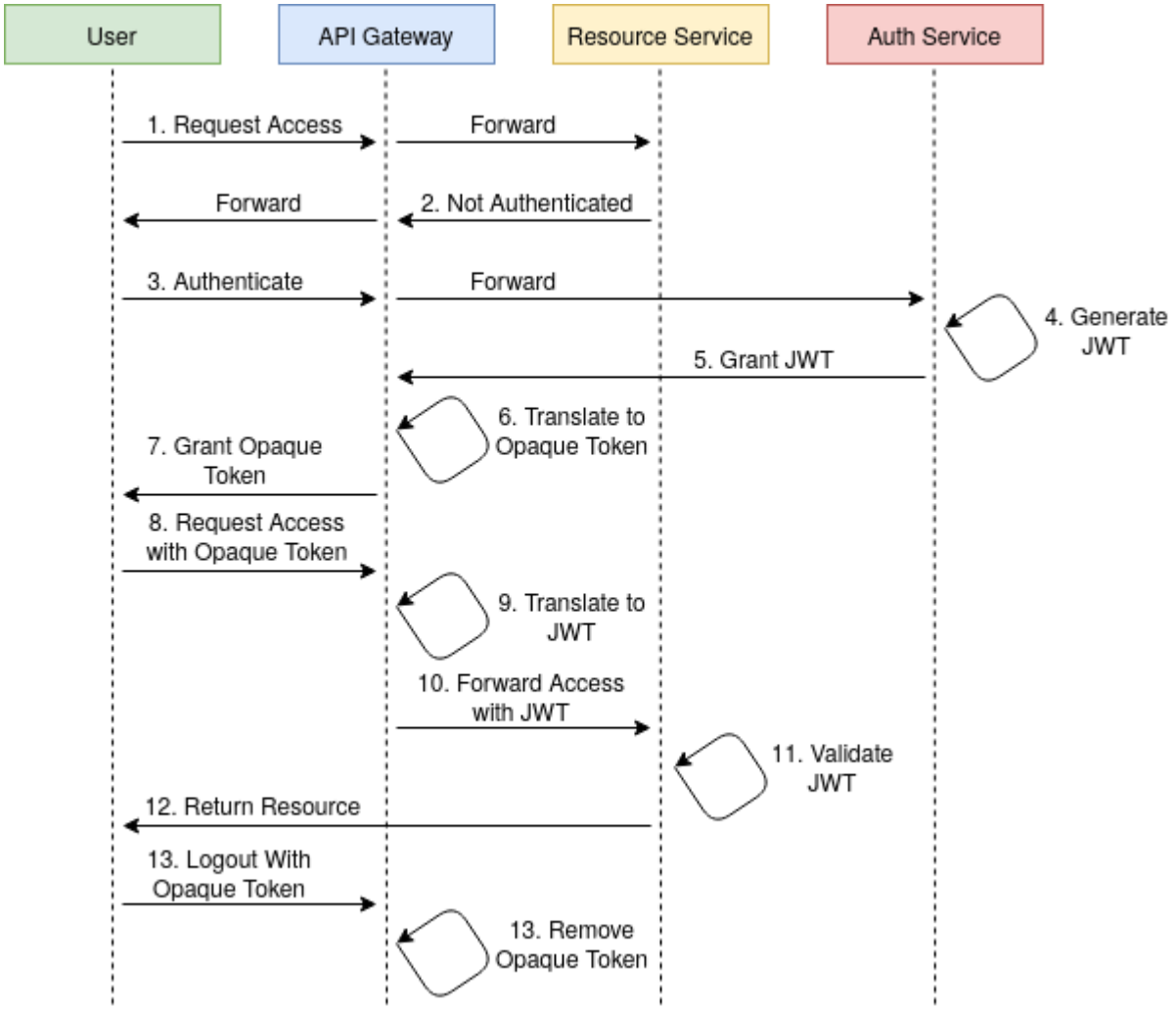

Figure 8: API Gateway and opaque token (He and Yang, 2017)

7 Cloud

7.1 Monolith

7.2 Microservices Architecture

8 Communication

As already explained, in MA, the service components can communicate using events, procedure calls or other methods available within a single server machine. Usually all this communication stays within a single computer and thus does not easily compromise confidentiality. This is not the case in MSA.

In MSA single services communicate via a network. There are multiple protocols or messaging system to choose from such as, REST API, Advanced Message Queuing

Protocol (AMQP), Enterprise Service Bus (ESB), and Remote Procedure Calls (RPC). For a more complete list view Yarygina (2018). In this paper REST API is to be examined further. In few instances some of the other systems are discussed briefly.

Miranda (2018) lists the typical fallacies that developers fall victim to when designing distributed systems:

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero
- The network is homogeneous

8.1 Representational State Transfer (REST)

Fielding (2000) presented REST in 2000. REST has become a very successful architectural style. The style was derived using various constraints, one of which is stateless communication. This entails that a request must contain all the information needed to fulfill the request because the server does not keep track of the client. All session state is stored in the client, of which the server has no prior knowledge before a request. In her doctoral thesis, Yarygina (2018) critiques the REST paradigm from the security perspective. She states that the design of the architecture does not meet the security requirements for web applications. She also states that REST does not allow for any server side sessions and thus token revocation is impossible. Tokens can be validated only for the correct issuer by signature and for expiration. As such tokens are more compatible with REST, but there still has to be the public keys in the server for signature verification.

8.2 Coping With Failure in Communication

Montesi and Weber (2016) present widely used design pattern for MSA. The Circuit Breaker can be used to mitigate the very likely case that a microservice operates slower than the other services calling it and runs out ot resources to fulfill the requests in time.

The circuit breaker is either implemented in the microservice or as a proxy between the client and the microservice. When the microservice does not service requests as intended, the circuit breaker trips and sends a failure message to the clients immediately when requests are received, thus allowing the microservice time to service the prior requests.

The circuit breakers can prevent an application from becoming completely unresponsive and crashing when a denial of service attack is carried out on the service.

9 Service Mesh

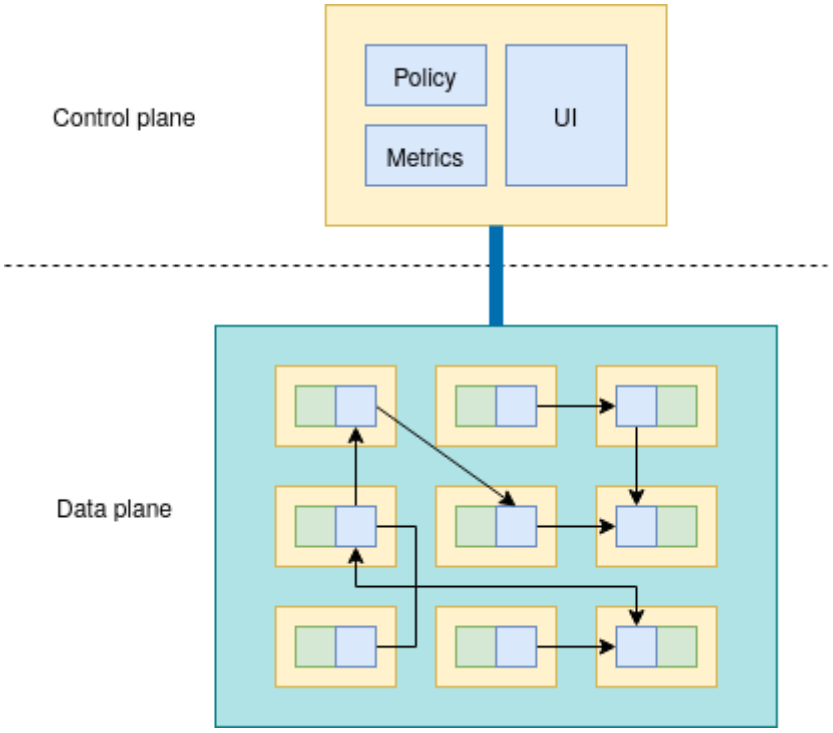

Figure 9: Service mesh basic architecture (Miranda, 2018)

A service mesh tries to solve the service-to-service communication challenges and to allow for monitoring of the entire system. In figure 9 a very basic service mesh architecture is presented. The architecture consists of a Control plane and a Data plane. The Control plane offers: a user interface for system administrators, Policy Information Point (PIP) and Policy Decision Point (PDP), and collected metrics of the behaviour and actions in and of the system. The data moves in the dataplane. Each microservice is accompanied by a proxy. All requests and responses flow through the proxies and ar controlled by the Control plane. The microservices do not necessarily have to be aware of the proxy that is in the information path.

There are several products available for a service mesh, such as Istio, Linkerd, and Consul Connect to mention a few. As was the case with authentication and security in general

it is not advisable to implement security critical features if there exists an off-the-shelf alternative.

The use of service mesh greatly simplifies the microservice since the proxy can contain many of the features that would otherwise repeat in all of the services. The service mesh can provide a certificate authority for the communication between services. The proxies act on behalf of the service and all proxy-to-proxy communication can be encrypted on the transport layer using TLS (Calcote, 2018).

Service mesh allows for multicloud installations.

10 Orchestration solutions

Docker Swarm, Kubernetes (K8s), Azure, sandbox, virtualization

11 Other Security Concerns

11.1 Transaction

Transactions can be used when updating database constents to make sure that atomicity, consistency, isolation, and durability (ACID) (Haerder and Reuter, 1983) is followed. When using MSA according to the definition each of the micro services should contain or have access to it's own data i.e. database. In MA Transactions easy/easier. In MSA not. How to make sure that a previously single operation is carried out in it's entirety when multiple services are used to perform the desired task?

11.2 Software Development

When software is developed using MA, it is usually deployed as a whole and the program code can be compiled, tested and used as a single unit or multiple modules. In contrast, a service implemented in MSA can be deployed in single microservice units, and thus each component can be worked upon individually and deployed once ready.

The immediacy in the deployment of the microservices entails a very specific security risk. Ahmadvand et al. (2018) present threats from malicious insiders working on the services as developers or other positions with access to sensitive information. In microservice development, the finished implementations are to be immediately released to production. There are steps in the CD pipeline prior to this but once tests pass in the test environments the pipeline is supposed to publish the changes to the actual production environment.

The paper presents four specific threats. The first one is that the knowledge of sensitive information is spread among the developers more widely than in MA. This is due to access needs by developers. The second threat is that the insiders monitoring and operating the running system intentionally harm the system by making malicious changes. The third threat is the developers knowing the configurations and their ability to make almost instantaneous changes to them or the microservices themselves. The last presented threat in the paper is the non-repudiation. The system is not able to disallow malicious requests when the developers have had access to the keys and other configurations. They can effectively implement services or requests that emit malicious requests or responds. Malicious attempts in a MA are more easily screened by performing security audits and by peer reviewing the code. In a MSA the knowledge of a single service and it's inner workings are shared by a more limited number of people. Finding the compromised actions from the interoperability of the distinct microservices is a daunting task.

Malicious attempts in a MA are more easily screened by performing security audits and by peer reviewing the code. In a MSA the knowledge of a single service and it's inner workings are shared by a more limited number of people. Finding the compromised actions from the interoperability of the distinct microservices is a daunting task.

11.3 Service discovery

Service discovery as presented in Montesi and Weber (2016) is a design pattern in which a registry is kept on currently running microservices. The microservices register them selves to the service discovery registry. This registry is used by either a router to route client service calls to running microservices or by the client directly.

11.4 Externalized configuration

To allow for easy configuration change management there should exist a configuration orchestration service. This service should have an API from which services in their startup can load their appropriate configuration. The configuration of the whole system can be easily maintained through the API.

The contents of the configuration is highly sensitive information. It consists of addressess, credentials and other information that alter the behaviour of the system. Therefore, the content must be stored safely and not allowed to be read or altered by unauthorized users.

11.5 Logging

Logging in MA is relatively easy. The chosen logging solution is used and logs are created in easily configurable locations.

In MSA this can be more difficult. Each of the microservices need to have their own logger and each of these need to be configured to log to a proper location.

11.6 Defence-in-Depth

Microservices can not trust any of the (Otterstad and Yarygina, 2017)

Jander et al. (2018) propose a solution for secure communication in MSA even in multicloud solutions. perimeter defence -> neglect security of individual microservices.

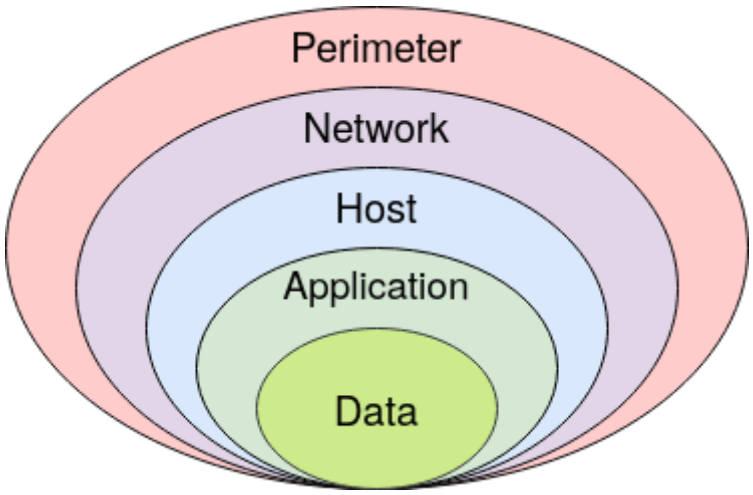

Figure 10: Defence-in-Depth

11.7 Testing

The possible ways to fail for a web service consisting of a large number of microservices are numerous. One solution for testing for outages is the Netflix created Chaos Monkey (cha, 2020). It is used for resilience testing and can terminate containers or virtual machines at random.

12 Conclusion

This paper discussed the security aspects of changing the architecture from MA to MSA.

In practice the previous MA that has been changed to MSA can have more aspects to go wrong than in previous MA implementation. The deployment can entail installing,

virtualization, monitoring, and a plethora of other tools. In some cases, these tools might not even exist and they have to implemented by inhouse developers and thus more costs are incurred upfront and also in the upkeep of the system. In addition to being more costly own development has higher security risks involved.

MSA has higher complexity due to more tools needed and having more potentially exposed attack surface. Security can be though of as being as good as its weakest link. In general, a MSA deployment has multiple layers which all have to be consistent and correct. One example is the configuration of the operating system on the server running the virtualization environment. All of the layers from the server hardware to the handling of errors in the actual code have to be of ample quality to mitigate a failure in security.

The communication that was in monolith a simple in-process call might not be possible as such in a MSA web service. The individual services communicate via the network with high overhead in comparison to a simple function call. Furthermore, the identity and authorization of the entity requesting an action or data can usually be trusted in an in-process call. The mechanisms to allow for proper authentication and authorization amount to even higher overhead for the MSA. There exists a very real risk for the development team to implement an insufficient security scheme.

In MSA the security should be implemented in depth. There must be a healthy mistrust on all requests and security should be built in to the system.

Security has to be taken into account right from the beginning of the project in which the architecture is to be changed. The choices made in the development of the web service when following a MA do not carry to the MSA as such.

Many security aspects were not discussed.

Future research on the security aspects of the different design patterns available for the architecture change.

References

Chaos Monkey, 2020. Available `https://github.com/Netflix/chaosmonkey`. Viewed 5.4.2020.

Welcome to OpenID Connect, 2020. Available `https://openid.net/specs/openid-connect-core-1_0.html`. Viewed 3.4.2020.

Mohsen Ahmadvand, Alexander Pretschner, Keith Ball and Daniel Eyring. Integrity protection against insiders in microservice-based infrastructures: From threats to a security framework. *Software Technologies: Applications and Foundations*, 2018.

Ross Andersson. Why information security is hard. *Univesity of Cambridge Computer Laboratory, Tech. Rep*, 2001.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland and Dave Thomas. Agile Manifesto, 2001. Available `https://agilemanifesto.org/`. Viewed 2.4.2020.

Lee Calcote. *The Enterprise Path to Service Mesh Architectures*. O'Reilly Media, Inc., 2018. ISBN 9781492041795. 1st edition.

Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2003. ISBN 9780321125217. 1st edition.

Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine, 2000.

M. Fowler and J. Lewis. Microservices - a definition of this new architectural term, 2014. Available `https://martinfowler.com/articles/microservices.html`. Viewed 15.2.2020.

T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *CM Computing Surveys (CSUR)*, 15(5):287–317, December 1983.

X. He and X. Yang. Authentication and authorization of end user in microservice architecture. *The 2017 International Conference on Cloud Technology and Communication Engineering (CTCE2017)*, 2017.

K. Jander, L. Braubach and A. Pokahr. Defense-in-depth and role authentication for microservice systems. *Procedia Computer Science*, 130(1):456–463, December 2018.

M. Jones, J. Bradley and N. Sakimura. Json web signature (jws). RFC 7515, RFC Editor, May 2015a. URL http://www.rfc-editor.org/rfc/rfc7515.txt. http://www.rfc-editor.org/rfc/rfc7515.txt.

M. Jones, J. Bradley and N. Sakimura. Json web token (jwt). RFC 7519, RFC Editor, May 2015b. URL http://www.rfc-editor.org/rfc/rfc7519.txt. http://www.rfc-editor.org/rfc/rfc7519.txt.

Miika Kalske, Niko Mäkitalo and Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. *Current Trends in Web Engineering*, Irene Garrigós and Manuel Wimmer, editors, pages 32–47, Cham, 2018. Springer International Publishing. ISBN 978-3-319-74433-9.

Tim McLean. Critical vulnerabilities in JSON Web Token libraries, 2015. Available https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/. Viewed 15.3.2020.

George Miranda. *The Service Mesh.* O'Reilly Media, Inc., 2018. ISBN 9781492031321. 1st edition.

Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and API gateways in microservices. *CoRR*, abs/1609.05830, 2016. URL http://arxiv.org/abs/1609.05830.

S. Newman. *Monolith to Microservices. Evolutionary patterns to transform your monolith.* O'Reilly Media, Inc., 2019. ISBN 9781492047841. 1st edition.

Christian Otterstad and Tetiana Yarygina. Low-Level Exploitation Mitigation by Diverse Microservices. *6th European Conference on Service-Oriented and Cloud Computing (ESOCC)*, Flavio De Paoli, Stefan Schulte and Einar Broch Johnsen, editors, volume LNCS-10465 of *Service-Oriented and Cloud Computing*, pages 49–56, Oslo, Norway, September 2017. Springer International Publishing. doi: 10.1007/978-3-319-67262-5_4. URL https://hal.inria.fr/hal-01677618. Part 2: Microservices and Containers.

Trygve Reenskaug. MVC XEROX PARC 1978-79, 2018. Available http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html. Viewed 29.3.2020.

Daniel Richter, Tim Neumann and Andreas Polze. Security considerations for microservice architectures. *Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,*, pages 608–615. INSTICC, SciTePress, 2018. ISBN 978-989-758-295-0. doi: 10.5220/0006791006080615.

Stack Overflow. Stack Overflow Developer Survey Results 2019. Available `https://insights.stackoverflow.com/survey/2019`. Viewed 1.2.2020.

T. Yarygina. Overcoming security challenges in microservice architectures. *Proceedings - 12th IEEE International Symposium on Service-Oriented System Engineering, SOSE 2018 and 9th International Workshop on Joint Cloud Computing, JCC 2018*, 2018.

M. Zari, H. Saiedian and M. Naeems. Understanding and reducing web delays. *in Computer*, 34(12):30–37, December 2001.

Verena. Zimmermann and Nina Gerber. The password is dead, long live the password – a laboratory study on user perceptions of authentication schemes. *International Journal of Human - Computer Studies*, 133(1):26–44, January 2020. doi: doi:10.1016/j.ijhcs.2019.08.006.