Service-to-Service Authentication in a Microservice Deployment

Benjamin Ellmer

BACHELORARBEIT

eingereicht am
Fachhochschul-Bachelorstudiengang

Mobile Computing

in Hagenberg

im Januar 2022

Advisor:

FH-Prof. DI Dr. Marc Kurz

Declaration

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

Hagenberg, January 31, 2022

Benjamin Ellmer

Contents

Abstract

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

Kurzfassung

An dieser Stelle steht eine Zusammenfassung der Arbeit, Umfang max. 1 Seite. ...

Chapter 1

Introduction

The trend towards highly-scalable software systems, like the microservice architecture emerged in the past years. The migration from the monolithic approach towards microservices has enormous consequences regarding the security of software systems [11]. Language-Level calls, which are calls within the same project migrate to remote calls over the network [3]. This offers a larger attack surface because intruders could spoof the communication among the services. Therefore authentication and confidentiality between services have to be implemented to secure the service-to-service communication. The confidentiality is usually provided using TLS, but the authentication requires the use of additional authentication mechanisms. The authentication is usually implemented using mutual TLS [4], but the authentication can also be provided using JSON Web Tokens. Therefore this thesis will describe and compare this authentication mechanisms to work out their differences, advantages and disadvantages.

1.1 Motivation

The International Data Corporation (IDC) has predicted that by 2022, 90% of all apps will feature microservice architectures [19]. So it is inevitable to deal with the numerous mechanisms to secure such systems properly. Authentication is one of the most important security challenges. Even if confidentiality is provided, when authentication is neglected intruders could perform attacks like the Man-in-the-middle-Attack to exploit the system. Such attacks could result in huge data leaks or could even allow attackers to abuse the system for their advantages.

The microservice architecture is based on having multiple services running on multiple servers. This results in a bigger attack surface, because multiple machines are exposed to the internet making it simpler to find vulnerabilities. This is one of the reasons why Netflix received massive attacks on their microservice based-systems in the past years [10].

This shows how vital microservice security is and the aim of this thesis is helping the microservice developers to choose the correct authentication mechanisms for their projects.

1.2 Challenges

Since the microservice architecture became as popular as in the last years, there is a lack of evaluation research and only limited insight into the particular security concerns, especially regarding service-to-service authentication. The most popular solutions and existing implementations are closed source, like the mTLS[1] Architecture of Netflix. Other freely available approaches are often poorly documented and therefore hard to understand [16]. The migration from the monolithic architecture to the microservice architecture results in a performance overhead [14]. Furthermore, the security measures produce high latencies because the services have to communicate to centralized authorities. So the mechanisms have to be implemented very efficiently, and caching should to be applied to minimize the resulting performance overhead. The microservice architecture and the service-to-service authentication bring some more challenges and difficulties, which will be declared and discussed in the following chapters.

1.3 Chapter Overview

[1] mutual Transport Layer Security is described in 3.3.3

Chapter 2

Microservice Architecture

This chapter introduces the microservice architecture concepts, which are necessary to understand why the later discussed approaches are needed. The principles used to design microservices lead to some characteristics, resulting in the motivations and challenges declared in this chapter. Furthermore some recommendations about the use cases of the microservice architecture are provided and the caused security consequences are declared.

2.1 Motivation

Companies like Netflix, Amazon, and Uber are front-runners for building software solutions using the microservice architecture [4]. The main idea is to split the business logic of an application into small autonomous services that work together. This means the programmers have to avoid the temptation of developing too large systems. This approach results in the following benefits [8]:

- Technology heterogeneity is achieved through the possibility to use different technology stacks for different services, depending on the needs of the services. It is even possible to use different data storage for the different microservices (e.g., graph database for users).
- Resilience is achieved since component failures can be isolated, so the rest of the system can carry on working by degrading the functionality of the system.
- Scaling is much more effective due to the possibility to scale only the parts of the system that really need scaling.
- The deployment is much more convenient because a single microservice can be deployed instead of deploying the whole application, even for small changes.
- The organizational alignment can be improved by assigning the work to small teams that work on smaller codebases, resulting in higher productivity.
- Composability is achieved, considering that the functionalities can be consumed in different ways for different purposes.
- Replaceability is optimized since rewriting a tiny service is much more manageable than replacing a few parts of a vast application.

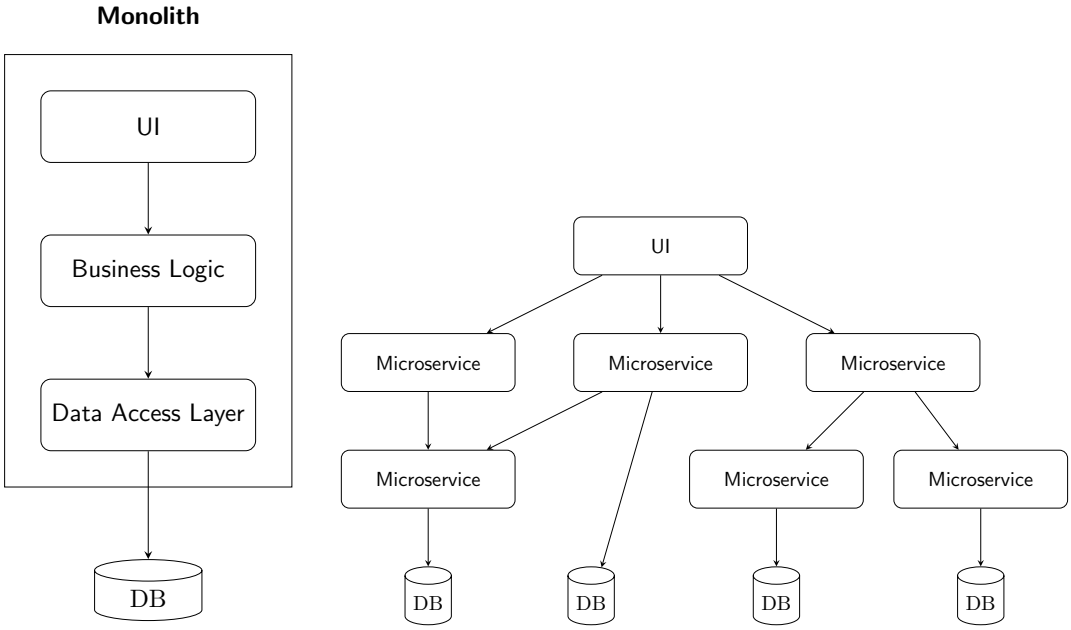

Figure 2.1: Example of monolithic architecture and a microservice architecture [5]

2.2 Comparison to the Monolithic Architecture

Figure 2.1 shows the architectural differences between monolithic and microservice applications. A monolithic application has a single unit containing the user interface layer (UI), the business logic layer, and the data access layer. Therefore it is much simpler to manage but brings the following downsides regarding the codebase [5]:

- New features and modifications of old features are harder to implement.
- Refactoring changes can reflect many parts of the software
- Code duplication raises since it is almost impossible to reuse existing code

The microservice architecture consists of multiple services focused on only one function of the business logic. Those services communicate with other services using remote calls (e.g., over HTTP), which causes higher latencies. Depending on the needs of the services, each service can have its own database, which can differ from the database system of the other services. It is also possible to share one database for multiple services, but this should be avoided to reduce coupling between the services.

2.3 Design Principles

It is hard to define principles, which will apply to all microservice architectures, but according to Newman, most of them will adhere to the following principles [8]:

Modelled around business concepts: The functionalities are structured around the business contexts instead of the technical concepts.

Adopting culture of automation: The microservice deployments embrace the culture of automation by using automated tests, continuous delivery, automated servers, and much more automation tools.

Hiding implementation details: The microservices hide as many implementation details as possible to avoid coupling. Especially the databases of the services should be hidden and can be accessed by other services using APIs.

Decentralising all The things: All approaches that could centralize business logic are avoided to keep associated data and logic within the service boundaries.

Independently deployed: The microservices should provide the possibility to deploy them without having to deploy any other service. Therefore the autonomy of the teams can be increased, and new features can be released faster.

Isolates failures: The microservices have to deal with misbehaving parts of the system and keep on providing as much functionality as possible, to prevent cascading failure.

Highly observable: It is not sufficient to observe a single service's behavior and status. Instead, the functioning of the whole system has to be monitored.

2.4 Challenges

There are some benefits of using the microservice architecture. It also introduces a set of challenges, which could argue to avoid the microservice architecture in some cases. According to Kalske et al. [5] the microservice architecture brings the following technical challenges with it:

- The declaration of the service boundaries is very hard [5], especially if the developers do not know the domain that well [8].
- The services should not become too fine-grained to prevent performance overhead. Otherwise, if they are not decomposed enough, changes to one service can affect multiple services.
- Continuous Delivery and Continuous Integration are necessary to manage the services and validate their functionality.
- The integration of the services into other services can become very hard due to the requirement to be available for all used technologies.
- Good logging mechanisms have to be used to recognize failures of microservices as soon as possible.
- Fault tolerance mechanisms have to be implemented to react to situations in which needed services do not respond.

The microservice architecture is gaining popularity, even if it produces so many challenges, showing how crucial its advantages are.

2.5 Usage Situation

According to Newman [8] it is better to start with a monolithic application if the architect does not fully understand the domain and has problems declaring the boundaries

for the services. In such cases, it is better to spend some time learning what the system does first and then break things down to microservices when the system is stable. Furthermore, Newman recommends eliminating legacy monitoring systems before splitting the application into more and more microservices. Otherwise, it could get very messy to stay in knowledge about the system's status.

Fowler [20] recommends considering microservices only when a system gets too complex to manage as a monolith. His essence is keeping the system simple enough to avoid the need of microservices since the microservice architecture can slow down the development considerably. The correlation between the development productivity and the complexity of a system comparing the microservice architecture with a monolithic architecture is shown in figure 2.2.

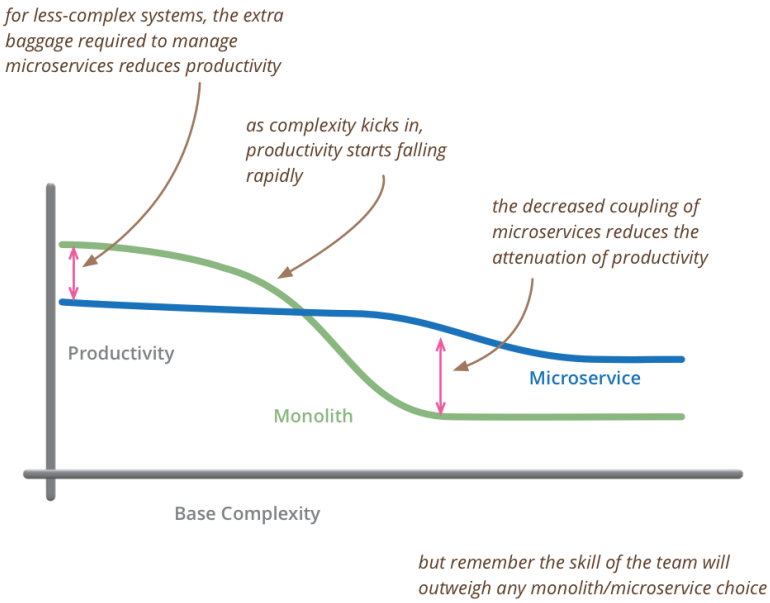

Figure 2.2: Correlation between base complexity and productivity in a microservice architecture compared to a monolithic architecture [20]

2.6 Security Consequences

The migration to the microservice architecture brings many consequences regarding the security of a deployment. Especially the network communications among the services introduce a set of vulnerabilities. Confidentiality, integrity, and availability have to be assured. The need for authentication mechanisms is a common issue of network security, but the migration from language-level calls to remote calls causes the need for authentication between microservices. Therefore, the authentication mechanisms discussed in this thesis are a consequence of the microservice architecture and can be neglected with a monolithic architecture.

Chapter 3

Related Work

This chapter summarizes the state-of-the-art concepts for microservice security and for public key based authentication. Furthermore the technologies, which are used to implement the later discussed authentication mechanisms are described. This chapter aims to give a good understanding of the whole domain and not only about service-to-service authentication.

3.1 Microserivce Security

Siriwardena and Dias [4] gave an exhensive guide of all topics related to microservice security. They separated the security of an microservice deployment into edge-level security and service-level security. The microservice architecture is based on separating the system in multiple parts. It is not sufficient to secure the system only on the edge-level or only on the service-level. Each part of the system has to be secured properly regarding confidentiality, authentication and authorization.

3.1.1 Edge-level security

Edge-level security is defined as the security mechanisms that protect the resources within the deployment from attackers outside the deployment. The API Gateway is responsible for the edge security it is the only entry point to the microservice deployment. All requests targeted for the APIs of the services are intercepted by the API Gateway. After validation of the requests, it dispatches the requests to the microservices. The main tasks of an API Gateway are authentication of the end user, authorization, and throttling. It authenticates the end user using access tokens, which come from access delegation technologies like OAuth 2.0 or Open ID Connect [12]. Due to outourcing the end user authentication to the API Gateway, the end user authentication has to be performed only once and not by every service. The API Gateway is able to perform simple authorization assertions. As soon as the authorization gets more granular, the microservices have to perform it on their own, because the API Gateway does not know the business logic.

3.1.2 Service-level security

Service-level security is defined as the security mechanisms that protect the communication among the microservices. According to Barabanov and Makrushin [1] service-level security can be decomposed into the sub functions service-level authentication, service-level authorization, and external identity propagation. Service-level security can either be implemented by the microservices themselves or by a service-mesh. A service mesh can be seen as a dedicated infrastructure layer, which manages the service-to-service communication of containerized services. In a typical microservice deployment with a service mesh, each microservice has its service proxy, which works transparently [4] The servce mesh is responsible for service discovery, routing, load balancing, traffic configuration, authentication authorization and monitoring [3].

Service-to-service-authentication

Authentication is the process of identifying the communication partner to protect a system from spoofing. Since the communication among microservices is done using remote calls, their communication has to provide authentication. Service-to-service authentication can be implemented in the following ways [4]:

- Trust the Network (TTN)
- mutual Transport Layer Security (mTLS)
- self signed JSON Web Tokens (JWTs)

Trust the Network is a security approach, that is based on the assertion that nobody has access to the components within a network perimeter. All components rely on the network security. Nevertheless, internal misbehaviour can lead to exploits allowing attackers intrude into the network perimeter and exploit the microservices [17]. Therefore the industry is heading towards zero-trust networks and the TTN approach became deprecated [4].

Service-to-servie authentication based on mTLS and self signed JWTs will be discussed in more detail in chapter 4.

Service-level authorization

Authorization defines the tasks that a principal is allowed to perform on a system. It requires that the principal is already authenticated because the authorization is performed based on the identity [12]. Service-level authorization gives the microservices more control to enforce access control. The authorization is usually performed using policy decision point (PDP) models like the centralized PDP model or the embedded PDP model [1, 4]. Proper service-to-service mechanisms are a precondition for service-to-service authorization, since the authorization can be bypassed with insufficient authentication [12].

External entity identity propagation

In order to perform the authorization correctly, the services have to know the context of the caller. The most popular technic for identity propagation is extracting the context of the user within JSON Web Tokens. The tokens are passed between the microservices and

the API Gateway. The propagated identity of the user can be extracted from the token, and the token's signature must be checked. The microservices can perform authorization based on the identity of the client [1, 4]. The chosen authentication mechanism do have an impact on the identity propagation, this will be discussed in more detail in chapter 4.

3.2 Public key-Based Authentication

Authentication can be performed in many ways. The authentication mechanisms which are later discussed in chapter 4 are both based on public key cryptography. Since public key cryptography provides higher security than symmetric cryptography, it is the preferred method for authentication mechanisms. Altough it requires higher computation and communication costs than symmetric cryptography [2].

3.2.1 Public key cryptography

3.2.2 PKI

3.2.3 Key Management

3.3 Technologies

3.3.1 X509.Certificate

X.509 certificates assure the users of a public key that the associated person or system owns the private key by binding public keys to subjects. Certificate authorities sign certificates and each communication partner who trusts the CA trusts the certificates signed by it. The most significant advantage of certificates is that they can be exchanged using untrusted communication channels because the signatures are not valid anymore when the contents of a certificate are changed. Therefore manipulations can be detected, and manipulated certificates can be declined [21].

Trust Path

When the client of a service wants to consume a service, which is hosted on a server, it has to obtain the server's certificate. If the client does not know the public key of the CA who signed the server's certificate, he has to obtain it. Obtaining the public key often results in chains because the client may have to work his way up until he reaches a CA he trusts. Such chains are also called certification paths. The way in which the clients can retrieve the CA certificates can be configured by the CA.

Fields

Depending on the version, a certificate can include more or less information. The information is always stored inside the tbsCertificate, signatureAlgorithm, and signatureValue fields and can be expanded using extensions.

The TBSCertificate contains the data of the certificate, including the following information:

- Subject of the certificate

Header. Payload. Signature

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. eyJzdWIiOiIxMjM0NTY3ODkiLCJpYXQiOjE1MTYyMzkwMjIsInVzZXJuYW1lIjoiYmVuamFtaW4uZWxsbWVyIiwiZW1haWwiOiJiZW5qYW1pbi5lbGxtZXJAeWFob28uY29tIiwiYWRtaW4iOmZhbHNlfQ. 0ksqN71oloNvq3IrY7w72uoTgPz9Gpn08p-KSbFulY0

Figure 3.1: Sample JSON Web Token

- Issuer of the certificate
- public key of the subject
- Validity period
- Additional information

The signatureAlogrithm field stores the information, which cryptographic algorithm was used to sign the certificate. Algorithms are declared by their identifier, the "OBJECT IDENTIFIER". The most commonly used algorithms are the RSA[1] algorithm and the Digital Signature Algorithm (DSA) [21].

The signatureValue field contains the value of the digital signature. It is obtained by signing the content of the tbsCertificate, using the algorithm specified in the signatureAlgorithm field. The signature is used to verify the validity of the information embedded in the tbsCertificate field.

3.3.2 JSON Web Token

A JSON Web Token (JWT) is a container, which can carry authentication and authorization assertions and further information in a cryptographically safe manner. An authentication assertion can be anything, which authenticates the user. Usually, usernames or e-mail addresses are used to identify a user uniquely. An authorization assertion can be any information about the access permissions of a user. For example, a JWT can include the information, whether the user is an admin or an unprivileged user [4].

Structure

A JWT is decomposed into the header, the payload, and the signature. The three parts are concatenated and separated by a dot [18]. A valid JWT could look like the JWT shown in figure 3.1.

The **header** contains the metadata related to the JWT, which is usually the type of the token and the signature algorithm. The specification defines that only HS256[2] and none algorithm must be implemented by conforming JWT implementation. It is recommended to additionally implement the algorithms RS256 and ES256[3] [18, 22]. The base64 encoded header is the first part of the JWT.

[1] Rivest Shamir Adleman
[2] HMAC SHA-256
[3] Elliptic Curve Digital Signature Algorithm (ECDSA) with 256-bit key

The **payload** is a set of registered and custom claims. A claim is a piece of information about an entity. The JWT specification defines registered claims, which are not mandatory for all cases but should provide a good starting point for a set of useful claims to ensure interoperability. Custom claims can be defined by the software architects, on their own, depending on their needs. The custom claims registered in the IANA registry are called public claims, and those not registered in the IANA registry are called private claims [18, 22]. The base64 encoded payload is the second part of the JWT.

The chosen signature algorithm signs the base64 encoded header, the base64 encoded payload, and a secret. The **signature** provides integrity for the message, and if it was signed with a private key, it additionally provides authentication [18]. The base64 encoded signature is the third part of the JWT.

3.3.3 Transport Layer Security

The Transport Layer Security (TLS) Protocol provides authentication, integrity, and confidentiality for the communication between two parties. It consists of two layers, the handshake protocol, and the record protocol [13].

mTLS

TLS itself is also called one-way TLS because it helps the client to identify the server, but not the server to identify the client. Therefore mutual TLS (mTLS) was introduced to provide authentication in both directions. The client and the server must own a private/public key pair, so it is more suited for the communication between two systems and not between users and servers [4].

Handshake Protocol

The handshake protocol is responsible for negotiating a cipher suite and for the authentication using X.509 certificates. The cipher suite declares the key exchange algorithm, the signature algorithm, the symmetric encryption algorithm, including the mode of the encryption algorithm and the hashing algorithm [7, 13]. The handshake varies on the key exchange method, but it can be separated into the following steps [6]:

1. The server and the client exchange Hello messages
2. The server sends its certificate to the client
3. The client sends a pre-master secret to the server and if mTLS is used, the client sends his certificate to the server
4. The client and the server finish the handshake, using the independently computed master secret

The steps of the handshake will be explained in more detail in chapter 4.1.1.

Record Protocol

The record protocol provides a secure channel for the communication between the parties. This is done by using the algorithms declared in the cipher suite. Confidentiality is assured, using symmetric encryption, and integrity is provided by Message Authentication Codes (MAC) [6, 7].

Chapter 4

Authentication Mechanisms

This chapter explains the concepts and details of the two compared authentication mechanisms. Only the mTLS approach and the authentication using self-signed JWTs approach are discussed in more detail since the Trust the Network (TTN) approach is deprecated and should not be used anymore [4].

4.1 Authentication based on mTLS

Mutual TLS is the most popular option for the service-to-service authentication of microservice deployments [4]. Securing the communication with TLS already provides integrity confidentiality and authenticates the server to the client. Since TLS does not provide authentication from the client to the server, it is insufficient for service-to-service security. Therefore mutual TLS is used, which provides an efficient and straightforward approach to authenticating the client to the server.

The authentication using mTLS requires a PKI, same as TLS on the internet. It is possible to use the already existing PKI of the internet, but this would make the key management much harder and would not bring any advantages. Therefore it is good practice to use a self-hosted PKI to have a root of trust within the network [4].

When mTLS is used, the server and the client must provide a valid certificate to create a communication channel. The issuer of the presented certificates must be trusted by all communicating parties [4]. If one communication partner does not have a valid certificate, the communication is neglected. Therefore each service needs its private key and the corresponding public key. Additionally, a signed certificate, which binds the public key to the certificate's subject, is needed. The certificates of the communication partners are exchanged during the TLS handshake.

This mechanism can also be used to authenticate the end-users of an application. The term Client Certificate Authentication (CCA) is used for this context [9]. Service-to-service authentication using mTLS is an implementation of CCA, but in this approach, the client is not the end-user, instead, it is another service.

4.1.1 Handshake

The TLS handshake is used to exchange the certificates of the participants and set up the connection. The handshake steps differ between the used algorithms and versions of

the TLS protocol. The following sequence and figure 4.1 should give an overview about the steps of the TLS handshake using mutual TLS [9]:

1. The client initializes the connection by sending a **ClientHello** message to the server. The **ClientHello** message includes a list of supported cipher suites, and the randomness, which is a combination of random bytes and the current date [23].
2. The server responds with a **ServerHello**, in which he chooses one cipher suite of the **ClientHello** message. Furthermore, the **ServerHello** contains the server's randomness.
3. The server sends the **Certificate** message, containing one or more certificates, which can be used to build the certificate chain. The client validates the sent certificates with his own trusted store. If it trusts the sent certificate chain, the handshake is continued.
4. The server sends the **CertificateRequest** message, in which the trusted CAs of the server are listed. The client can use this list to choose the correct certificate he has to present for the Client Certificate Authentication.
5. The server sends the **ServerHelloDone** message, signalizing that the server finished his Hello message.
6. The client responds with his **Certificate** message, which is similar to the servers **Certificate** message, but contains the client's certificate chain.
7. The client then generates a random value, the pre-master secret, which is later used to derive symmetric keys for the cryptographic operations defined in the cipher suite. The pre-master secret is then encrypted using the server's public key. Therefore, only the owner of the corresponding private key, the server, can decrypt this message. In the end, the encrypted pre-master secret is transferred to the server within the **ClientKeyExchange** message.
8. The client has to prove that he owns the corresponding private key of the certificate he sent. Therefore he has to encrypt the hash of all previous messages with his private key. This encrypted hash is then sent to the server within the **CertificateVerify** message. The server can decrypt the hash with the certificate's public key and can calculate the hash on its own to check whether the decrypted hash is correct or not.
9. The client sends a **ChangeCipherSpec** message to signal the server that all following messages will be protected with the protection mechanisms defined in the cipher suite.
10. The last message of the handshake is the **Finished** message, which is an encrypted hash of all previous messages.
11. Same as step 9, but from the server.
12. Same as step 10, but from the server.

After the TLS handshake, both participants know the secret, that is used to encrypt and decrypt messages. The handshake would have almost the same steps when mTLS is not used. Only the **CertificateRequest** message of the sever and the **Certificate** message and the **CertificateVerify** message of the client are unique for mTLS. One special case of the handshake is that the client responds to the **CertificateRequest**

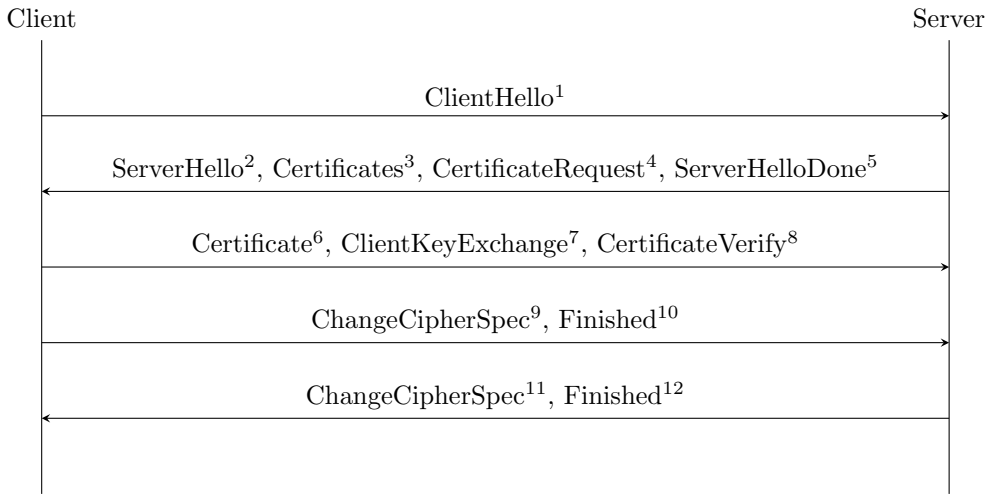

Figure 4.1: TLS handshake using mTLS [9]

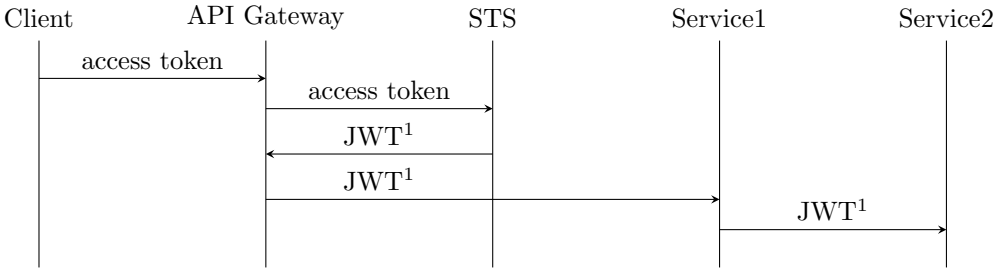

Figure 4.2: Use the same JWT for each request [4]

with an empty **Certificate** message. Depending on the configuration of the server, the connection without a certificate can be allowed or neglected [9].

4.1.2 Passing the end user context

For some functionalities, the identity of the microservice is not relevant. Instead, the identity of the end-user is relevant. In those cases, the microservices have to pass the end-user context when they consume the logic of other microservices. The most popular approach for passing the end-user context with mTLS is JSON Web Tokens. The JWTs can be embedded within the HTTP Request body or using URL parameters. This approach can be implemented in multiple ways [4].

Generally, the user obtains an access token from any token service. This token could be an OAuth2, OpenID Connect, or any other token. The user has to send this token with each request. The token is then validated by a Security Token Service (STS). If the token is valid, the STS returns a JWT, which can then be used to consume other services. When one microservice calls another microservice, he sends the JWT, and if this microservice has to consume another microservice, he also passes the same token to the next microservice [4]. The workflow of this approach is shown in figure 4.2

Another approach is that the STS is used to generate a new token for each request like it is shown in figure 4.3. When the STS generates a new token for each request, he fully knows all performed requests. Therefore the STS could implement further authorization logic [4]. Nevertheless, the frequent calls could result in an bigger workload for the STS and decrease the system's performance.

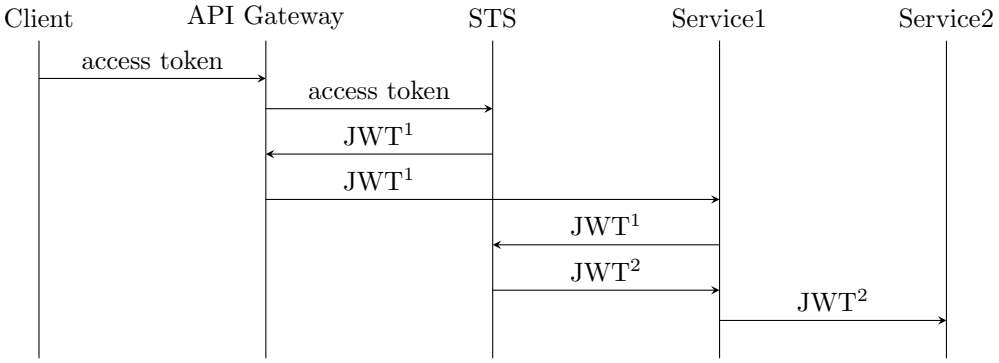

Figure 4.3: Generate new JWT for each request [4]

Both approaches result in some overhead, especially the approach in which a new JWT is needed for each request. Additionally, when the microservices are located in different trust domains, those approaches can get very complicated since a microservice only trusts the STS within its trust domain [4]. The problem with the JWTs is that they can be stolen and used by intruders. Therefore it is necessary to use mTLS together with JWT.

4.1.3 Conclusion

mTLS is an efficient mechanism to implement service-to-service authentication. Since the communication between the services is usually done using HTTPS, the services already use TLS. Therefore, mTLS does not cause much configuration overhead, and no new technologies are necessary [4].

One crucial advantage of the TLS handshake is that the private keys are never exchanged, and the session keys are always different due to the usage of the randomness. This means that even if an intruder can get the session key of a communication channel, he cannot use this key for another session. Furthermore, it is not possible to retrieve any information about the private key of the communication partners with the knowledge of the session key. This shows how secure mTLS is, even for advanced attacks [9].

From the developer's perspective, mTLS does not require much implementation logic. The service which acts as the server has to be configured to use certificate authentication. Depending on the used technologies, this is usually done by setting a few configuration parameters in the code or directly on the webserver. The service which acts as the client has to be configured to send his certificate during the TLS handshake. Most HTTP Client libraries support simply attaching the certificate to each HTTP request. Nevertheless, the developers do not have to implement much logic resulting in the problem that they do not have much control over the system. The developers

have to rely on the implementation of the webserver developers. This means that if a web server has security-related bugs, the microservice developers can not solve them independently. For example, the apache webserver, one of the most popular webservers, had many issues in combination with CCA. Arnis Parsovs [9] researched the problems of the apache webserver and gave an extensive guide on how to circumvent all bugs when CCA is configured.

The biggest challenge of mTLS is the key management, which was described in more detail in the chapter 3.2.3. Key management is responsible for key provisioning, key revocation, key rotation, and more management tasks. Usually, the key management requires a self-hosted PKI for the deployment. For small applications, the key management can be kept very simple. Nevertheless, as soon as the deployment grows and many services are running simultaneously, automation tools are required. Therefore the management overhead of mTLS is much harder to handle than the implementation of mTLS itself [4].

The previously mentioned challenges and motivation result in the conclusion that mTLS is a beneficial and efficient approach when the developers do not require to fully control each aspect of the authentication. Especially when the end-user context has to be shared among the services, mTLS might not be the most efficient solution. Even if mTLS may not be the ultimate tool for all security challenges, when the requirement is service-to-service authentication, mTLS does its job, and it does it well. This is the reason why mTLS is the most popular approach for service-to-service authentication.

4.2 Authentication based on self-signed JWTs

Self-signed JWTs can be used to provide authentication for service-to-service communication. Same as mTLS, it is based on asymmetric cryptography, and each service needs to own a key pair. The main idea is that the sender creates a JWT, which is signed with his private key. The receiving service can then check the signature of the JWT with the public key of the sending service. The signed JWT is transferred within the Authorization header of the HTTP request [4]. Since the JWT does not have any fixed structure, it is possible to embed contextual data like the user context as claims within the JWT. Therefore parameters and information about the user do not have to be passed within the body or as URL parameters [4].

The usage of self-signed JWTs does not provide any confidentiality for the communication among the services. Therefore TLS should be used to provide confidentiality for the service-to-service communication. Since the authentication is not dependent on TLS, it is possible to use other mechanisms instead of TLS. For example, JSON Web Encryption or any other encryption mechanism can be used, but usually, it does not make sense to exchange TLS [4].

The usage of self-signed JWTs additionally provides non-reputability. This means each action is bound to the service that created the JWT, and the service can not repudiate that the JWT was created by him [4]. Whenever non-reputability is a requirement of the system, self-signed JWTs are the superior authentication mechanism.

4.2.1 Non Repudability

Digital signatures can be used to achieve non-reputability cryptographically. Nonrepudability binds the actions performed by a service, to the service or API Gateway who initiated the action. In general the recipient of a message is provided with a proove of the origin of the message. Therefore the recipient is protected against situations in which a sender denies that the sent a message [15].

Nonrepudability and authentication are very similar. Authentication is about convincing the other party that an event is valid. With nonrepudiation, it is even possible to prove the truth of an event to a third party [15].

A practical example in which non-reputability is useful would be an online shop. Assuming that the shop has an **OrderService** and a **PaymentService**. When the payment is sucessfully accomplished using the functionalities of the **PaymentService**, it has to signalize the **OrderService**, that the order was paied. Therefore the **PaymentService** would send a request to the **OrderService** containing a JWT signed using the private key of the **PaymentService**. When the **OrderService** stores the JWT and the request, it can always proove that the request was originated by the **PaymentService**. It could not be an other service, because only the **PaymentService** can create a valid signature for the public key of the **PaymentService**.

4.2.2 Passing the end-user context

The approaches to how the end-user context is passed between the services are similar to the approaches explained in 4.1.2. The big difference is how the JWT is transferred among the services. While with mTLS, the JWT was embedded within the body or as a URL parameter, with self-signed JWTs, it can be embedded within the JWT that already has to be transferred. This is done by appending a nested JWT within the claims of the self-signed JWT. The signature of the JWT, which is used for authentication, can still be verified by using the service's public key. The signature of the nested JWT is verified using the public key of the STS. This means the JWT is carried in a way that can not be forged. Therefore it is better to use self-signed JWTs for the authentication when the end-user context is relevant in many situations [4].

4.2.3 Conclusion

Service-to-service authentication using self-signed JWTs is not only a mechanism for authentication. It additionally provides nonrepudiation and makes sharing the user context more convenient. Otherwise, the implementation of self-signed JWTs is much more challenging than the implementation of mTLS. It is insufficient to configure the webserver correctly and append a certificate to each request. Every service has to know how to encode and decode JWTs. Furthermore, to achieve nonrepudiation, all received JWTs must be stored for an adequate timespan, requiring additional database storage.

One major advantage of the authentication using self-signed JWTs is, that the developers can vary the implementation depending on their needs. For example the developers can vary the technology used to provide confidentiality, even if this might not be necessary in most cases, the possibility can be an advantage [4]. Furthermore the developers can define additional parameters, which have to be provided within the transferred

JWTs. This freedom of choice is not guaranteed with mTLS, because it is a strictly defined protocol and has very few configuration options.

Sadly the biggest challenge of mTLS, which is the key management, can not be avoided using self-signed JWTs, because it also requires each service to have its key pair. Nevertheless, the key management can be simplified since it is unnecessary to have a CA responsible for signing each certificate. Still, regarding the webserver setup, it makes sense to have one superior authority, that is, the root of trust and whose chained certificates are trusted.

As a result, self-signed JWTs are the preferred authentication mechanism when the target is to achieve nonrepudiation or when the system is very dependent on sharing the user context. Nevertheless, this leads to additional implementation overhead and requires developers who are specialized in security-related systems.

Chapter 5

Project Structure

This chapter aims to show the concrete structure of an project, which implements the previously discussed authentication mechanisms. The focus of this chapter is showing the effects of the different authentication mechanisms to the project itself. For the simplicity topics like authorization, key management and user context sharing are not handled in this chapter. The visualizations of this chapter are based on the backend of a flea market app.

5.1 Flea Market App

The flea market app is an Android App written in the programming language Kotlin. The main features are buying, renting and swapping items. The user is authenticated using firebase authentication. He has to present his access token to the Microsoft API Gateway with each request. The backend of the app is based on the microservice architecture. The microservice are mainly implemented in C# using the ASP.Net Core framework. Actually the backend contains the following services:

- AdService
- UserService
- ChatService
- MediaService
- SubscriptionService
- ReportService

Each service has its own PostgreSQL database. The services are hosted on Microsoft Azure using docker containers. Only the AdService and the UserService will be used to show an exemplary example of the communication within the deployment when the discussed authentication mechanisms are implemented.

5.2 Exemplary Example

Figure 5.1 shows an extract of the previously described backend and visualizes the communication of the needed components. In this example the user fetches all ads which are in his surrounding. Therefore the AdService is used to retrieve the ads from

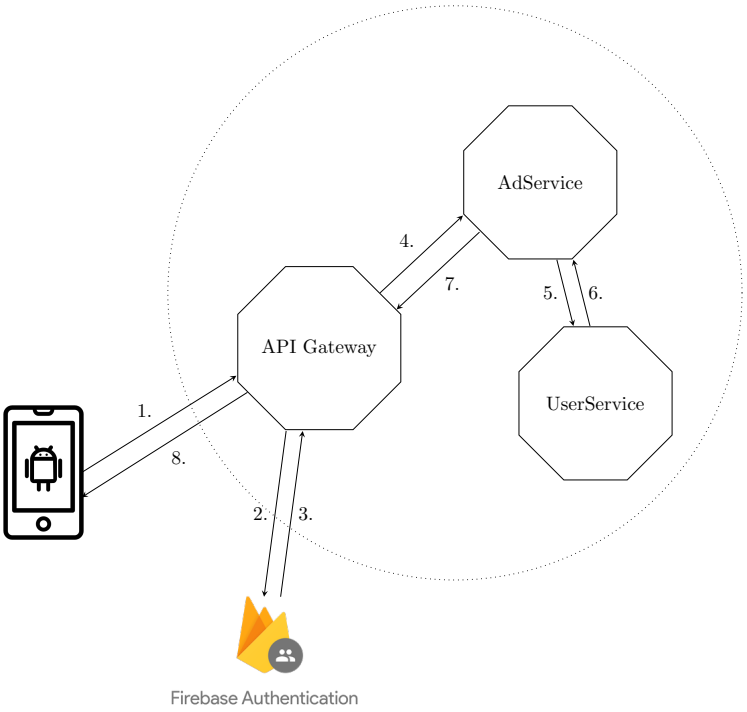

Figure 5.1: Communication among the components of the exemplary example

the database and the UserService is needed to preview information about the seller. In this use case the following components are necessary:

Android App: The Android App is the User Interface for the client to access the functionalities of the services. The requests sent by the Android App are sent in beyond of the user.

Firebase Authentication: The Firebase Authentication service is responsible for validating the access tokens wich are transferred by the users. The app also communicates with this service to get an access token, but this is done in an earlier stage.

API Gateway: The API Gateway is the only entry point into the deployment. Therefore the API Gateway is the only component which directly communicates to the Android App.

AdService: The AdService is responsible to manage all ads offered to the user of the app.

UserService: The AdService is the service which is responsible for managing the users.

5.2.1 Workflow using mTLS

1. The client sends an API request to the Microsoft API Gateway using the Android App. The communication between the API Gateway and the App is secured using HTTPS. Therefore the server is authenticated to the client using TLS. The client

is authenticated to the server by embedding an firebase access token within the Authorization header.

2. The API Gateway sends the received token to the Firebase Authentication service to validate it. This procedure is defined as a policy within the policy file of the API Gateway.
3. The Firebase Authentication service returns the information whether the persented token is valid or not.
4. When the client provided a valid access token, the request of the client is forwarded to the **AdService**. The communication between the API Gateway and the **AdService** is secured using mTLS. This means both parties have to present a certificate, signed by a trusted CA. Therefore the webserver of the **AdService** has to be configured to allow or even require Client Certificates.
5. After processing the request from the API Gateway the **AdService** sends a request to the **UserService** to retrieve the additional information about the owner. The communication between the **UserService** and the **AdService** is also protected using mTLS, so it has to be configured like the **AdService**. The certificate is appened to the request using a HTTPClient, which is injected to the Controller of the WebService using Dependency Injection.
6. When the **UserService** processed the request, it responds with the expected result. The response does not require to perform the authentication again, since TCP connection between the **AdService** and the **UserService** is still opened.
7. After the **AdService** processed the response from the **UserService**, it uses the opened connection with the API Gateway and transfers its response.
8. The API Gateway forwards the response from the **AdService** to the Android App. This connection is still secured using TLS and not mTLS. The App can now process the response and present the requested information to the user.

5.2.2 Workflow using JWT

The workflow using JWT and the workflow using mTLS are very similar, therefore only the steps which differ between the mechanisms are described again.

4. The request from the client is forwarded to the **UserService**. Therefore the API Gateway has to create a valid JWT to communicate with the **UserService**. On the Microsoft API Gateway, the logic to create a JWT is implemented as a policy. The JWT, which is signed using the private key of the API Gateway is transferred within the Authorization header.
5. The **AdService** has to retrieve additional information from the **UserService**. Now the **AdService** has to present his JWT to the **UserService**. This is done by using a HTTPClientFactory, which automatically embeds the signed JWT within the Authorization header, when the request is sent. If the **AdService** does not know the certificate of the **UserService**, it will deny the request and ask the **AdService** to present its certificate. In this case the request is repeated and the certificate of the **AdService** is transferred during the TLS handshake, like it would be transferred using mTLS.

5.3 Implementation Details

5.3.1 mTLS

As already discussed in chapter 4, the implementation of mutual TLS does not require very much logic on the service-side. The best way to implement mTLS, in a ASP.Net Core API is using the **Microsoft.AspNetCore.Authentication.Certificate** library. It provides the mechanism to add certificate authentication as **AuthenticationScheme** and implement custom **CertificateAuthenticationEvents**. The **CertificateAuthorityService**, implements the interface **ICertificateAuthorityService** which is shown in listing 5.1. It provides features to manage a custom certificate truststore and validate certificates using this truststore. An instance of the **CertificateAuthorityService** is injected to the **CertificateAuthenticationEvents**. The dependcy injection is managed by the **Microsoft.Extensions.DependencyInjection** library. The **UseAuthentication** and **UseAuthorization** functions of the **ApplicationBuilder** have to be called to use the created **AuthenticationScheme**. The functions of the API Controllers have to be marked with the [**Authorize**] annotation, to require the client to be authenticated to consume the function. Therefore it is possible to exclude some functionalities from the certificate authentication.

```
1 public interface ICertificateAuthorityService {
2   public void AppendCertificate(X509Certificate2 certificate);
3   public bool ValidateCertificate(X509Certificate2 certificate);
4 }
```

Listing 5.1: ICertificateAuthorityService interface, which is implemented by the injected CertificateAuthorityService

5.3.2 JWT

The best way to implement the authentication using self-signed JWTs in ASP.Net Core is a custom middleware. A custom middleware provides the possibility to perform actions on the receiving request before, they are handled by the API Controllers. The **JWTAuthenticationMiddleware** validates the JWT and the certificates of all received request. The validation is performed like it is shown in 5.1.

When a client certificate is transferred within the TLS handshake, it is checked using a service that implements the **ICertificateAuthorityService**, which was shown in listing 5.1. If the certificate is valid, it is mapped to the issuer of the certificate within a custom truststore of the **JWTValidationService**. The **JWTValidationService** implements the **ITokenValidationService** interface which is shown in listing 5.2. Both the **CertificateAuthorityService** and the **JWTValidationService** are injected into the **Invoke** function of the **JWTAuthenticationMiddleware** using dependecy injection.

```
1 public interface ITokenValidationService {
2   public bool PutIntoTruststore(X509Certificate2 certificate);
3   public bool ContainsIssuerInTruststore(String issuer);
4   public bool ValidateTokenWithTruststore(String token, string issuer);
5   public bool ValidateTokenWithTruststore(String token);
6 }
```

Listing 5.2: ITokenValidationService interface, which is implemented by the injected JWTValidationService

Algorithm 5.1: Pseudocode of the request validation using self-signed JWTs

```
certificate ← request.Connection.ClientCertificate
token ← request.Headers[Authorization]
missingCertificate ← false
if token is null then
    response.StatusCode ← 403Forbidden
    authenticated ← false
else
    issuer ← null
    if certificate is not null then
        if certificate is trusted then
            if putCertificateIntoTruststore(certificate) is successful then
                issuer ← certificate.subject
    else
        if containsIssuerInTruststore(token.issuer) then
            issuer ← token.issuer
        else
            missingCertificate ← true
    if issuer is not null && validateJWT(token) is valid then
        authenticated ← true
    else
        if missingCertificate is true then
            response.StatusCode ← 401Unauthorized
        else
            response.StatusCode ← 403Forbidden
        authenticated ← false
```

References

Literature

[1] Alexander Barabanov and Denis Makrushin. "Authentication and authorization in microservice-based systems: survey of architecture patterns". *arXiv preprint arXiv:2009.02114* (2020) (cit. on pp. 8, 9).

[2] An Braeken. "Public key versus symmetric key cryptography in client–server authentication protocols". *International Journal of Information Security* 21.1 (2022), pp. 103–114. URL: https://doi.org/10.1007/s10207-021-00543-w (cit. on p. 9).

[3] Ramaswamy Chandramouli. "Microservices-based Application Systems". *NIST Special Publication* 800 (2019), p. 204 (cit. on pp. 1, 8).

[4] Wajjakkara Kankanamge Anthony Nuwan Dias and Prabath Siriwardena. *Microservices Security in Action*. Simon and Schuster, 2020 (cit. on pp. 1, 3, 7–12, 14–17).

[5] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. "Challenges when moving from monolith to microservice architecture". In: *International Conference on Web Engineering*. Springer. 2017, pp. 32–47 (cit. on pp. 4, 5).

[6] Hugo Krawczyk, Kenneth G Paterson, and Hoeteck Wee. "On the security of the TLS protocol: A systematic analysis". In: *Annual Cryptology Conference*. Springer. 2013, pp. 429–448 (cit. on p. 11).

[7] Aleksandr Kurbatov et al. "Design and implementation of secure communication between microservices" (2021) (cit. on p. 11).

[8] Sam Newman. *Building microservices*. "O'Reilly Media, Inc.", 2021 (cit. on pp. 3–5).

[9] Arnis Parsovs. "Practical issues with TLS client certificate authentication". *Cryptology ePrint Archive* (2013) (cit. on pp. 12–16).

[10] Anelis Pereira-Vale et al. "Security mechanisms used in microservices-based systems: A systematic mapping". In: *2019 XLV Latin American Computing Conference (CLEI)*. IEEE. 2019, pp. 01–10 (cit. on p. 1).

[11] Ekaterina Shmeleva et al. "How Microservices are Changing the Security Landscape" (2020) (cit. on p. 1).

[12] Prabath Siriwardena. *Advanced API Security*. Springer, 2014 (cit. on pp. 7, 8).

[13] Sean Turner. “Transport Layer Security”. *IEEE Internet Computing* 18.6 (2014), pp. 60–63 (cit. on p. 11).

[14] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. “Workload characterization for microservices”. In: *2016 IEEE international symposium on workload characterization (IISWC)*. IEEE. 2016, pp. 1–10 (cit. on p. 2).

[15] Wei Wu et al. “How to achieve non-repudiation of origin with privacy protection in cloud computing”. *Journal of Computer and System Sciences* 79.8 (2013), pp. 1200–1213. URL: https://www.sciencedirect.com/science/article/pii/S0022000013000640 (cit. on p. 17).

[16] Tetiana Yarygina and Anya Helene Bagge. “Overcoming security challenges in microservice architectures”. In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE. 2018, pp. 11–20 (cit. on p. 2).

[17] Zirak Zaheer et al. “Eztrust: Network-independent zero-trust perimeterization for microservices”. In: *Proceedings of the 2019 ACM Symposium on SDN Research*. 2019, pp. 49–61 (cit. on p. 8).

Online sources

[18] Auth0. *JWT Documentation*. URL: https://jwt.io (visited on 10/27/2021) (cit. on pp. 10, 11).

[19] International Data Corporation. *IDC FutureScape: Worldwide IT Industry 2019 Predictions*. Oct. 2018. URL: https://www.idc.com/research/viewtoc.jsp?containerId=US44403818 (visited on 10/22/2021) (cit. on p. 1).

[20] Martin Fowler. *Miroservice Premium*. May 2015. URL: https://martinfowler.com/bliki/MicroservicePremium.html (visited on 11/29/2010) (cit. on p. 6).

[21] *RFC2459*. URL: hhttps://www.ietf.org/rfc/rfc2459 (visited on 12/29/2021) (cit. on pp. 9, 10).

[22] *RFC7519*. URL: https://datatracker.ietf.org/doc/html/rfc7519 (visited on 10/27/2021) (cit. on pp. 10, 11).

[23] *The Beauty of SSL-Handshake*. URL: https://medium.com/@vraghuvaran123/the-beauty-of-ssl-handshake-4286afa543cf (visited on 01/24/2022) (cit. on p. 13).