

The positioning of the OAuth2 client in a Microservice-based Architecture

A comparison between the implementation of the OAuth2 client in the Gateway and in the
frontend

Bachelor Thesis

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science in Engineering

to the University of Applied Sciences FH Campus Wien

Bachelor Degree Program: Computer Science and Digital Communications

Author:

Ursula Rauch

Student identification number:

00514397

Supervisor:

Leon Freudenthaler, BSc MSc

Date:

!!!FEHLT NOCH!!!

Declaration of authorship:

I declare that this Bachelor Thesis has been written by myself. I have not used any other than the listed sources, nor have I received any unauthorized help.

I hereby certify that I have not submitted this Bachelor Thesis in any form (to a reviewer for assessment) either in Austria or abroad.

Furthermore, I assure that the (printed and electronic) copies I have submitted are identical.

Date: 15.01.2023

Signature:

Abstract

This thesis investigates different implementations of Authorization and Authentication with OpenID Connect (OIDC) and OAuth 2.0 (OAuth2) in a microservice architecture (MSA) environment...

Kurzfassung

Diese Arbeit untersucht unterschiedliche Implementierungen von OpenID Connect (OIDC) bzw. OAuth 2.0 (OAuth2) im Kontext von Microservice-Architekturen (MSA) ...

List of Abbreviations

!!!ALT! NICHT gebrauchte raushaun! BCP	Best Current Practice
CRUD	Create Read Update Delete
ECDSA	Elliptic Curve Digital Signature Algorithm
ES256	ECDSA SHA-256
HMAC	Hash-based Message Authentication Code
HS256	HMAC SHA-256
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
JOSE	JavaScript Object Signing and Encryption
JSON	JavaScript Object Notation
JWE	JSON Web Encryption
JWS	JSON Web Signature
JWT	JSON Web Token
MSA	Microservice Architecture
mTLS	mutual Transport Layer Security
OIDC	OpenID Connect
RFC	Request for Comments
OP	OIDC Provider
POC	Proof of Concept
PoLP	Principle of least privilege
RSA	Rivest-Shamir-Adelman
SHA	Secure Hash Algorithm
SSO	Single Sign-on
XACML	Extensible Access Control Markup Language

Acknowledgement

macht man das hier nicht? wenn doch, an welcher stelle genau?

Key Terms

Authentication

Authorization

BFF

Gateway

JWT

Microservice Architecture

OAuth 2

OpenID Connect

Contents

1	Introduction	1
1.1	Related Work	1
1.2	Microservice-based vs Monolithic Architecture	1
1.3	Authentication and Authorization	1
1.3.1	Authentication	2
1.3.2	Authorization	2
1.3.3	The role of authentication and authorization in MSA	2
1.4	OAuth 2 and OpenID Connect	3
1.4.1	OAuth2	3
1.4.2	The Access Token	4
1.4.3	OpenID Connect (OIDC)	7
1.5	The positioning of the OAuth2 client in a MSA system	7
1.6	Methodology	8
2	Implementation	9
2.1	The Teapot - High level design	9
2.2	Setup with Spring Boot and Keycloak	11
2.2.1	Spring and Spring Boot	11
2.2.2	Keycloak Server	13
2.2.3	Spring Cloud Gateway as OIDC Client	15
2.2.4	The Resource Server	19
2.2.5	Role mapping with Keycloak and Spring Boot Resource Server	23
2.3	Load testing with JMeter	28
2.4	Code Analysis	28
3	Discussion and Results	29
3.1	Response times	29
4	Conclusion and Future Work	30
	Bibliography	31
	List of Figures	31
	List of Code Listings	31
	List of Tables	33

1 Introduction

!!! Einleitung allgemein, Forschungsfragen:

Die Rolle des Gateways für Authentifizierung und Autorisierung mit OAuth2 und OpenID Connect in Microservice-basierten Architekturen. Das Gateway kann sowohl als OAuth2-Client, als auch als Resource Server implementiert werden. Im ersten Fall muss das Gateway als Client einen Access Token vom Authorization Server beantragen und diesen an den Resource Server, also einen dahinter liegenden Service weiterschicken. Wenn das Gateway selbst als Resource Server implementiert ist, muss das Frontend als Client fungieren und den Access Token beschaffen. // Forschungsfrage: Achtung, neue Forschungsfrage!

1.1 Related Work

1.2 Microservice-based vs Monolithic Architecture

Was ist MSA

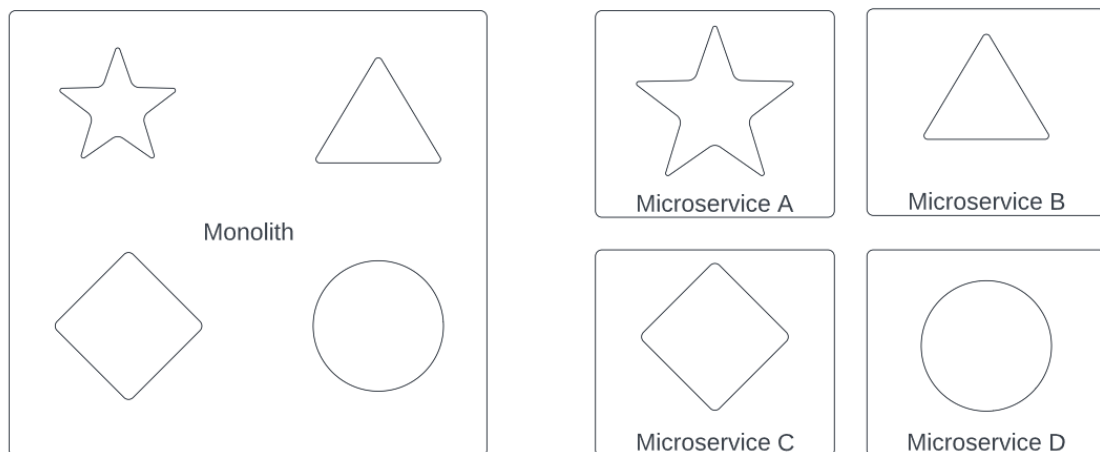


Figure 1.1: A very abstract illustration of a monolith and microservices according to [?]

1.3 Authentication and Authorization

Authentication and Authorization are related concepts and often they only appear linked to each other, which might be the reason why the distinction between the two is not always so clear. This section gives a short introduction and explains how they are different from each other.

1.3.1 Authentication

Authentication deals with the question of identity. Often it is crucial to know the identity of a user in order to decide if the user is allowed to access a specific resource. If this is the case it might also be important to know later, that this person has accessed the resource, for example if someone has misused their right to access the resource for another than the allowed purpose, like stolen or manipulated data. The process of authentication involves the information who someone is, for example during a login process this can be a user id, and also the prove that this information is correct. This prove can be in the form of something only this person knows, like a secret password, or something only this person has, like a code that can be sent to this persons phone number, or some unique attribute of this person, like biometric data [?, pp. 59ff]. A combination of those is multifactor authentication and increases the level of security. Not only human persons need authentication, but also systems can possess a kind of id that identifies them and a prove in the form of a secret code, a token or a certificate. In any case, there must be a database that can be consulted to verify that this prove is valid, otherwise it would be of no value, but it is not necessary for the system that controls access to a resource to possess the database itself, it can delegate the whole business of user authentication to a different entity that functions as an identity provider (IP).

1.3.2 Authorization

Authorization on the other hand is about permissions. Naturally, in order to decide if someone should be authorized to access a resource, this will often require knowing who this person is. But in theory this is not necessarily the case by definition. A person might earn the right to access a resource regardless of who they are, or they might be authorized because of certain attributes, as it is the case for example with many public toilets that are open only for certain genders. It doesn't matter at all what a person's name is or when they were born, but they are allowed to access the toilet resource only if they appear to have the correct gender attribute. But in other more complex systems, especially where security is a priority, authentication is crucial in the combination with authorization. Roles or attributes of persons or other systems are determined based on their identity and in a second step it can be verified if the role or attribute authorizes them to access the resource. A person with the intention to access a resource might be able to prove that they are who they say they are, but they might still not be authorized to access the resource. This too requires the consulting of data to know who is allowed to do what.

1.3.3 The role of authentication and authorization in MSA

With the characteristics of MSA, authentication and authorization play an even more important role than ever, although Service oriented Architecture (SOA) and distributed systems already point in the same direction [?]. The OWASP Top 10 API Security Risks [?] for 2023 list again broken authentication and broken authorization on top. The main security challenges related to MSA were identified by Dragoni et al. to be a large surface attack area, network complexity, trust and heterogeneity [?]. While heterogeneity and especially trust might not always present to the same degree in all MSA systems, the large surface attack area and network complexity are inherent features in any MSA system compared to a monolithic architecture. Other authors have pointed out the necessity (and also the increasing adoption in the industry) of a zero trust policy for MSA, which means that each microservice considers all other microservices or actors as potentially hostile and therefore no service should trust any incoming request without verifying it's integrity, regardless of who

the sender might be [?], [?]. Compared to a monolithic system, this makes the implementation and management of authentication and authorization in a MSA more complex. While from an end user's perspective the experience might not be different when they communicate with the MSA system via an API gateway, which hides the underlying complexity of the system. There are only a couple of entry points relevant to them [?]. But inside the system, the situation is very different: Because each single service must expose at least one endpoint to be of use in the MSA system, the number of endpoints that have to be protected is at least equal to the number of services.

1.4 OAuth 2 and OpenID Connect

To begin with, OAuth 2 (OAuth2) is a framework or open protocol for authorization only. OAuth2 is considered the unofficial standard for securing the access to APIs [?, p. 81], [?]. It does not deal with authentication, but authorization only. It has been described in Request for Comments (RFC) 6749 [?] and RFC 6750 [?], followed by a long tail of further specifications [?], [?], [?], [?], Security Considerations [?], [?] and a series of Security Best Current Practice Internet drafts [?]¹. The original specifications from 2012 have been updated by RFC 8252, "OAuth 2.0 for Native Apps" [?] and by "OAuth 2.0 for Browser-Based Apps" [?]. Since this is not even a comprehensive list, it is understandable that the new specification for OAuth 2.1 [?], which has been published just recently, has been long awaited by some. It replaces RFC 6749 and 6750.

Because OAuth2 is not intended to be used for authentication, OpenID connect (OIDC)[?] was created as a separate layer for this purpose, in addition to OAuth2 in 2014. Therefore it should be clear that these two are not alternative concepts to choose from, but when securing services, both should be used.

1.4.1 OAuth2

The motivation for the creation of OAuth2 was to enable a client application to access a resource on a server on behalf of the owner of that resource without the need to pass on the resource owners credentials to this client for authentication [?]. Third-party access to resources is a very common practice, the standard example is an application that needs access to a user's facebook timeline [p. 81]siriwardenaAdvancedAPISecurity2020, but it may as well be a different resource like pictures or a person's calendar. Sharing a password with a third-party application is undesirable for several reasons [?] : Passwords are inherently weak, especially in combination with human users, who tend to reuse passwords on different unrelated systems and servers would have to implement support for password authentication. Third-party applications would store these passwords, typically in clear-text. By authenticating with the server as the user, the third party application would have access to all of the user's data with the same permissions as the user themselves. The only way to revoke access for such an application would be to change the password, which would naturally exclude all other third-party applications as well.

OAuth2 separates the role of the *client* (the third-party application) from the *resource owner* (the end user who allows access to a resource to the client), so that the client is not required anymore to pretend to be the user by using the user's credentials when communicating with the *resource server* (the server holding the resource). Instead of a password the client sends an *access token* with every request to the resource server. This access token has a limited lifespan and other attributes that allow to control the extend to which the

¹This is not a comprehensive list.

client can access the resources on that server. This access token is issued to the client by an *authorization server* after the user has given consent for the client to access the resource. Following the principle of separation of concerns, the authorization server is also the only one dealing with authentication of the user `hardtoOAuthAuthorizationFramework2023`.

The explicit consent of the resource owner for the client to access the resource is called *authorization grant*. The client can use this grant to obtain the access token from the authorization server. There are several different authentication grant flows defined for OAuth2. The preferred grant type for most use cases is the *Authorization Code* grant type². Other grant types are the *Refresh Token* grant and the *Client Credentials* grant. With the *Authorization Code* grant the authorization server issues first only a code to the user agent together with the redirect url. After the authentication, the user is sent back to the client by the authorization server with this redirect url and the authorization code. The client application can now send the code to a different endpoint at the authorization server, the `token endpoint`, and exchange it for an access token. In this way the access token is transmitted only via backchannel communication, which makes it harder for attackers to intercept the token. There are also additional measurements to make the code exchange more secure, like the `state` parameter and *Proof Key for Code Exchange* (PKCE). When the client redirects the user agent to the authorization server (this is called the authorization request), it must create a code challenge and add it to the request, unless it is a confidential client and the OIDC nonce value is used. Other required parameters are the `response_type` (code for the authorization code grant) and the `client_id` [?]. Figure 1.2 shows the steps involved between the different roles during the authorization code grant flow.

1.4.2 The Access Token

The OAuth2 specification does not define the nature of the access token. Although it can be any arbitrary string that has no further encoded information (this would be a reference token), it is considered best practice to use self-contained tokens like JSON Web Tokens (JWT), because the resource server itself can validate them and determine if the authorization is sufficient for the request, without having to build up a connection to the authorization server each time or maintaining a token database [?]. Specifically, in the context of distributed systems like MSA, the use of JWT as access tokens is advisable. One reason is that they can be validated locally and not every single request to the system has to be validated first with the authorization server, which could result in performance loss [?]. The drawback of local token validation is that if a token has been revoked, the validating service doesn't know this and will give the client with the token access until it expires [?]. The use of JWTs as access tokens for OAuth2 is specified in RFC 9086 [?].

The client requests access tokens from the `token endpoint` at the authorization server with a POST request including at least the `client_id` and the `grant_type` [?]. Confidential clients, which are capable of maintaining secret information will also include a form of authentication, symmetric (password) or asymmetric (a signed JWT or mTLS). It is also possible for the client to include a `scope` request parameter. This is necessary for example when using OIDC (see section 1.4.3).

The response from the authorization server, if the authorization request was valid, includes several parameters [?]: the freshly issued access token for the client and `token_type`, which is usually "Bearer", as well as a `scope` parameter are required by the specification. The

²Before OAuth 2.1 there was also the *Implicit* grant and the *Resource Owner Password Credentials* grant, but they are not considered secure anymore. Their use has already been excluded by OAuth 2.0 Security Best Practice documents[?] and they are completely omitted in the new specification `hardtoOAuthAuthorizationFramework2023` (See also [?]).

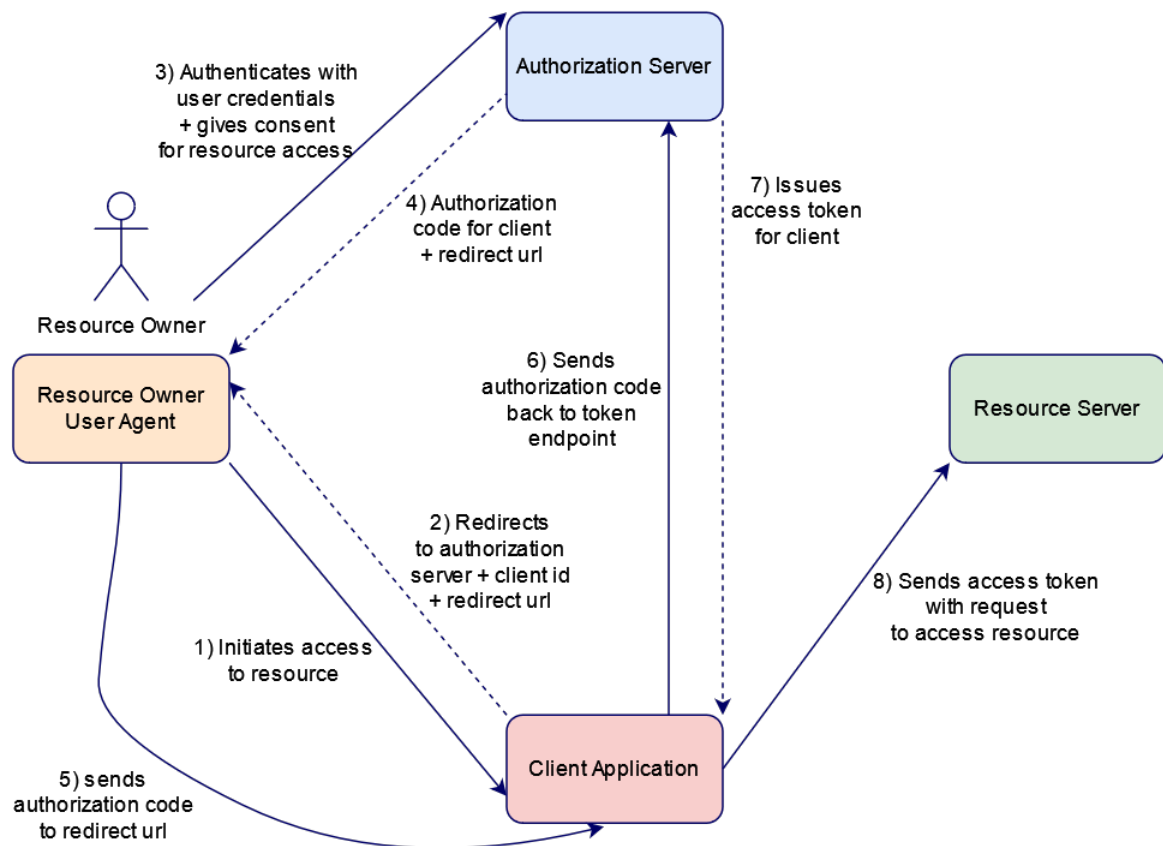


Figure 1.2: Steps of the authorization code grant in the interaction between all roles as described in [?]

1 Introduction

```

1 "access_token":
2     "eyJhbGciOiJSUzI1NiIsInR5cCIgOjAiSldUIiwia2lkKiA6ICJHVGVZ5eG9HVTBjbGxob2hnX3U1SGcwaWwudmVudmMUGzS2xXN1BQTRBn0.eyJleHAiOjE2ODc5NzUxNTMsImhhbmhkdCI6MTY4NDk3NDk3MywiYWxpbnRpIjoiaGR0cDovL2hvc3QuZG9ja2VybmludGVybmFsOjEwMDAxL3JlYWxtcy90ZWFWb3QiLCZzdWIiOiIzMmMuU1NjQzYyOTNWU5LTQ5NDYtOTJjMC04Nzd1MWEzNTNlZjcilCj0eXAiOiJCZWVyZXIiLCJhenAiOiJ0ZWFWb3QtZ2F0ZXdsesIsInblnc3Npb25fc3RhduGUioiI4MDA1NGEyYS1jZWRjLTRlOGMtOTNmY0zNmI2ZjY1ZDY4NWUiLCJhy3IiOiIxIiwiaWF0IjE5YXNxb3dlZC1vcmlnaW5zIjpbIi0iXSicmVhbGgiYWNjZXNzIjpb7InJvbGVzIjpbIm9mZmxpbmVfYWNjZXNzIiwidW1hX2F1dGhvcm16YXRpb24iLCJkZWZhdWx0LXJvbnGVzLXRLYXBvdCI6InRlYV91c2VyIl19LCJyZXNvdXJfZV9hY2Nlc3MiOnsidGVhcG90LWdhduGV3YXkiOnsicm9sZXMiOlscihJpdmlsZWdlZF91c2VyIiwidXNlcjdfX0sInnjb3BlIjojdGVhX3JlYWQiLCJzaWQioiI4MDA1NGEyYS1jZWRjLTRlOGMtOTNmY0zNmI2ZjY1ZDY4NWUifQ.vbU10Esac38vl7hhoMFGD-CN7JwQGVU6Ff2uzn7ocAgqycfShcidZG3J0X3FFM191XdC7akc3PGR957ouTplwNUgrjplZ1lQmvIk06xCPferh-x6VAX_px083hbRqCyKJ5m5fKBey_dnuhB9vLn0IYRD7YR7eGHHi49zIcWLNdxsichVjmlHaZOV_jzl05MEGGjQbsi5ybViKXShl13lLC2u3NNpa6SdnjqydWhdqVktKx8e-TLQ8xTCLlvDKB8FULVs4_3dMRZo3HxpqbKbn6ixZZY_P3-ucqV2KZ5HPcti0zX5FWiFuk-mJ3VzPdW04K1mPIZRRA16K0Gxjt2PW",
3 "expires_in": 180,
4 "refresh_expires_in": 0,
5 "token_type": "Bearer",
6 "not-before-policy": 0,
7 "session_state": "80054a2a-cedc-4e8c-93e3-36b6f65d685e",

```

Figure 1.3: Token response from the Keycloak server in Postman

authorization server can decide if the scope specified in the request will be included. Further recommended is `expires_in`. A refresh token can also be included in the response, depending on the type of client and other factors. Figure 1.3 shows a token response from Keycloak. The lifespan for the access token is configured to be three minutes and because the authenticated user has only read privilege, the scope `tea_read` is included.

A JWT is base64-encoded and is made of three parts, the header, payload and the signature. Any JWT that is used as access token for OAuth2 must be signed, as specified in RFC 9068, to ensure it's integrity, preferably with asymmetric cryptography [?]. The header contains the alg parameter indicating the signing algorithm, which has to include RS256, and the typ parameter, which has to be at+jwt (recommended) or application/at+jwt. The payload contains the claims set where the following claims are mandatory for OAuth2 access tokens [?]:

- `iss`: Issuer. Indicates the issuer of the access token.
- `exp`: Expiration time. The life span of the access token before it becomes invalid.
- `aud`: Audience. Indicates the resource for which the access token should be used.
- `sub`: Subject. The subject of the access token is an id that belongs to the resource owner, if involved, otherwise the client.
- `client_id`: Client ID. Identifier for the client requesting the access token.
- `iat`: Issued at. The time when the access token was issued.
- `jti`: JWT ID. Unique identifier for the JWT.

An example for a decoded access token is shown in figure 2.14, although it does not conform entirely to the specifications, as the `typ` header parameter contains `JWT` instead of the required `at+jwt` value and the `aud` claim as well as the `client_id` claim are missing. The client id is instead present with the `azp` claim.

When validating the access token, the resource server checks the signature, lifespan, scope and possibly other authorization parameters for the specific resource [?].

1.4.3 OpenID Connect (OIDC)

When it comes to authentication of end-users, OIDC [?] gives the answers that were left out bei OAuth2. The content of an OAuth2 access token is of no interest for the client. When the client needs information about a user's identity, it can request an additional *ID token* by adding `openid` to the scope parameter in the authorization request, which thus becomes a authentication request. Also the `redirect_uri` parameter is now required. Among other optional parameters there is the `nonce` parameter to mention. It is a string value, used to protect against replay attacks, that should be unguessable and it represents session state between the client session and the ID token. The ID token will then be issued by the authorization server which is now called OIDC provider (OP), together with the access token and delivered as a part of the token response [?]. The ID token itself is a signed JWT which contains the `iss`, `sub`, `aud`, `exp` and `iat` claims as a requirement. When using the authorization code flow, also the `at_hash` claim is required. It contains a the base64 encoded left most half of the corresponding access token's hash. The `nonce` claim is required if it was present in the authentication request, with the same value, which will be verified by the client. Other optional claims are `amr` (authentication method references), `acr` (authentication context class reference) and `azp` (authorized party), containing the client id of the authorized client. The ID token itself does not contain personal user information. Instead there is a separate OAuth2 protected `/userinfo` endpoint where the client can request additional metadata about the user with the access token [?].

1.5 The positioning of the OAuth2 client in a MSA system

!!! nochmal anschauen. wo soll der abschnitt hin und was ist die intention? Titel? When bringing the two concepts of MSA and OAuth2 together, some decisions have to be made about the role of each service, which also depend on the overall architecture of the microservice system. Many different architectural patterns for MSA have been proposed and studied [?]. One very common pattern is the API Gateway, which functions as entry point for requests coming from a browser, a mobile application or any other kind of frontend [?], so that the single backend services are not required anymore to be reachable from the outside. A very similar pattern is Backends for Frontends (BFF), where as many different gateway services are implemented as there are different frontend applications [?]. The distinction between the two patterns seems to be fluent to a certain degree. Sometimes it is called a BFF when an API Gateway assumes the functionalities of an OAuth2 client [?], [?], however this does not seem to meet the full definition of a BFF.

The most simple implementation would probably be to implement only the gateway as OAuth2 resource server, but not the backend services, assuming that the gateway will deal with any unauthorized request. For this scenario there are other ways to secure the communication between services, like mutual Transport Layer Security (mTLS), supposedly the most popular option (see [?, pp. 137ff]). This means that the service holding a requested resource might not necessarily be a resource server in the sense of OAuth2. The Gateway

will decide if the user should access a resource and it will forward the request only if the user can prove to have the necessary authorization (access token). However this comes with drawbacks, for example it does not meet the requirement of defense in depth, where access control happens on several layers and access control in one single point can become hard to manage with a more complex access policy and many roles involved [?]. For better security it is also possible to implement several resource servers in a series, which either renew the access token or hand down the original access token to the next downstream service after validation [?, pp. 161ff]. This is similar to the chain of responsibility pattern [?]. A simplified version of this concept has been implemented and tested for this thesis (see chapter 2). In both cases however, the job position of a (registered) OIDC client is still vacant. There should be a service or application that is able to determine if the user has already authenticated, refer to the OP for authentication and obtain the access token on the user's behalf. This means that any possible frontend has to implement OIDC client functionality.

Another version, which can sometimes be found in OAuth2/OIDC MSA tutorials, relieves the frontend of this burden and implements the gateway directly with the functionality of an OAuth2 client. This variant was also implemented for this thesis and is described in detail in section 2. The services beyond are resource servers, so that the communication between the API gateway and the next downstream service is secured via access tokens, while the gateway keeps a session for the communication with the frontend to know if the requesting user is authenticated.

Naturally, both variants, with the API gateway or the frontend application as the OAuth2 client bring advantages and disadvantages in different aspects, like system performance, code complexity and further security considerations. In chapter 2, not only the implementation of a prototype is described, but also the setup for a performance test in order to compare response times of both variants. The results of these tests are presented and discussed in section 3 (!!! discussion separat oder getrennt?). (!!! code complexity und security?)

1.6 Methodology

!!! Darf ich das so schreiben? The first research question, how OAuth2 authorization and authentication with OIDC are implemented in MSA using Spring Boot and the Spring Security Framework with a Keycloak server, was answered mainly by collecting and combining information from online tutorials and use them to implement a prototype as a proof of concept. The implementation is described in detail in chapter 2. It should be noted that the aim was not to follow or develop a best practice model. For example in production TLS would be required on all connections, but was left out because the focus lies on authentication and authorization of users with OAuth2 and OIDC. Also mTLS was considered to be beyond the scope of this thesis and therefore left out. Correct authentication and authorization of end users was tested and examined using Postman³ and Wireshark⁴.

In the first version, the API gateway was implemented as OAuth2 client. In order to investigate the matter of the positioning of the OAuth2 client, the system was recreated in three versions, containing only the necessary features for a performance comparison between the API gateway as client, as resource server and a completely unsecured version to relate the outcome. Performance tests were carried out using Apache JMeter⁵. Details about the setup are documented in section 2.3. (!!! code complexity und security?)

³<https://www.postman.com/>

⁴<https://www.wireshark.org/>

⁵<https://jmeter.apache.org/>

2 Implementation

All implementations and testing were carried out on a Lenovo Thinkpad T490s with a Intel®Core™i78665U CPU, 1.9 GHz Base Frequency, 4,8 GHz Max Turbo Frequency and 4 Cores. The operating system was Microsoft Windows 11 Pro.

All Services including the Keycloak server were deployed with Docker Compose version 2.17.3 and the Docker version was 23.0.5.

2.1 The Teapot - High level design

!!! In order to become familiar with MSA, OAuth2 and OIDC, the first project that was built is a virtual tea kitchen, called "The Teapot". It then served as a starting point for the comparison of different OAuth2 client positions, with some simplifications and changes to serve the purpose.

In the original Teapot system the user can view a list of available types of tea and make a cup with the chosen tea. The backend is a MSA and consists of the API Gateway, the Tea Service with a MongoDB database, which offers endpoints for creating or updating a type of tea, requesting the list of all available types, deleting tea and "making tea", where the user gets back a message containing the requested type of tea or just hot water, if the requested tea is not available. There is also a separate Milk Service and a Eureka Discovery Service where the Gateway and the other Services are registered. The gateway offers endpoints to the outside world and stands between the other services and the users. It routes requests requests to the Tea and Milk Service respectively, so that the user or any frontend doesn't have to communicate directly to the services beyond the gateway. A keycloak server is deployed for security, serving as both, identity server for user authentication and authorization server for the services. The high-level architecture of the teapot is depicted in figure 2.1

However, since there is a lot of functionality present that is not necessary for this research, the whole system was rebuilt in a even simpler version: All that we need is the gateway and one additional service for the gateway to communicate with, and of course the keycloak server. So the Milk Service as well as the Discovery Service disappeared completely. The database still exists in the new system, but since it became clear that it would only add unnecessary overhead to the requests it is not in use anymore. Neither is the whole create/read/update/delete (CRUD) functionality. Instead, the Gateway and the Tea Service offer "hello"-endpoints that were used in the beginning for debugging. In the end, these endpoints were used for load testing, as will be described in more detail in section ???. They return a simple string message and do not require the database. This means that the gateway has two relevant endpoints: /helloauth, which the gateway itself responds to immediately, and /hellotea/name, which is routed from the gateway to the Tea Service. name can be any string and will be returned in the responding message. The remaining, stripped-down system is represented by figure 2.2.

In total, there are three versions of this system: the first version where the Gateway acts as the OAuth2 client and the Tea Service serves as the resource server, the second version where both the Gateway and the Tea Service function as resource servers, and a third version with no security implementation at all. The second version would require the inclusion of a

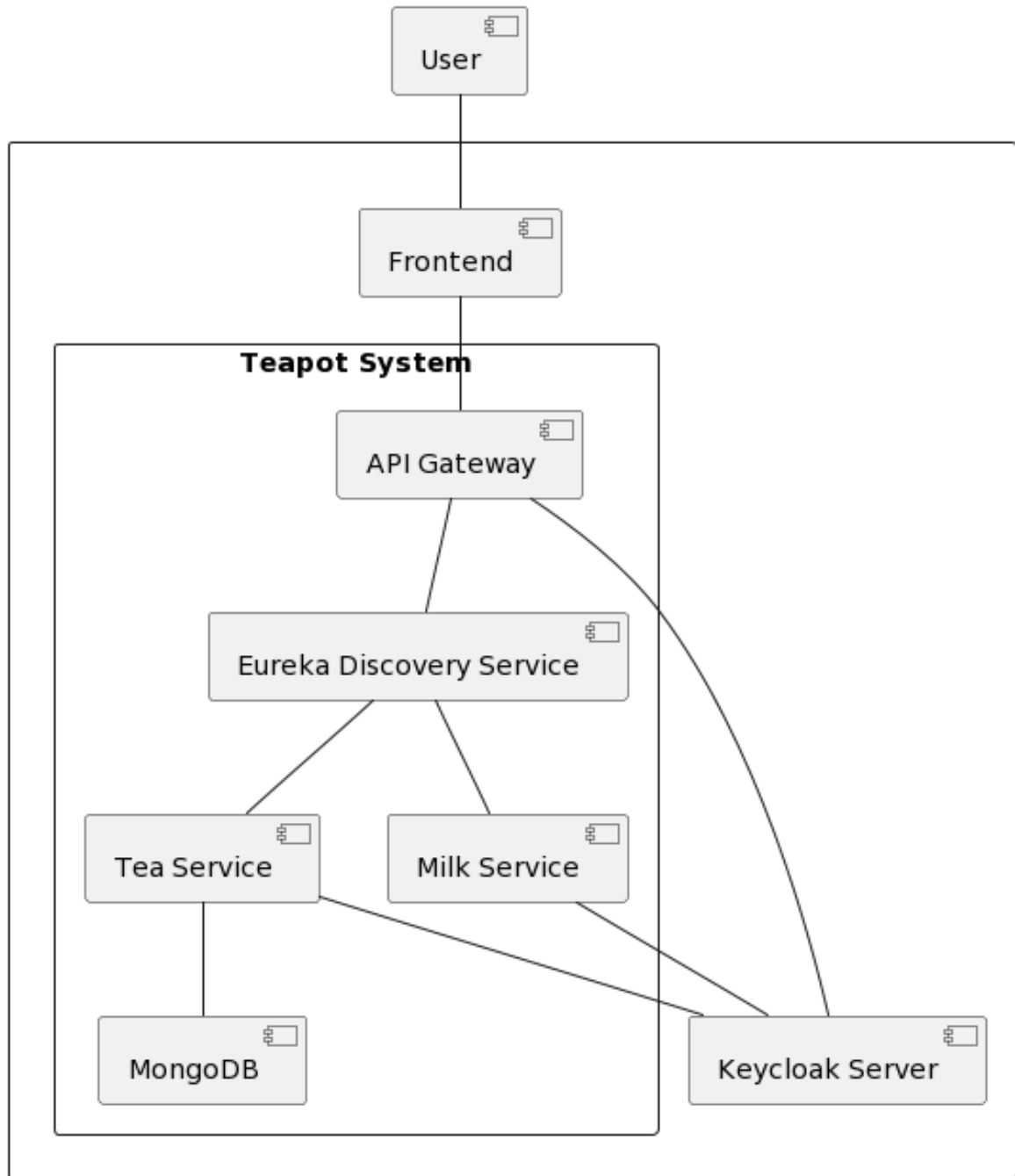


Figure 2.1: High level diagram of the implemented services and their relation to each other

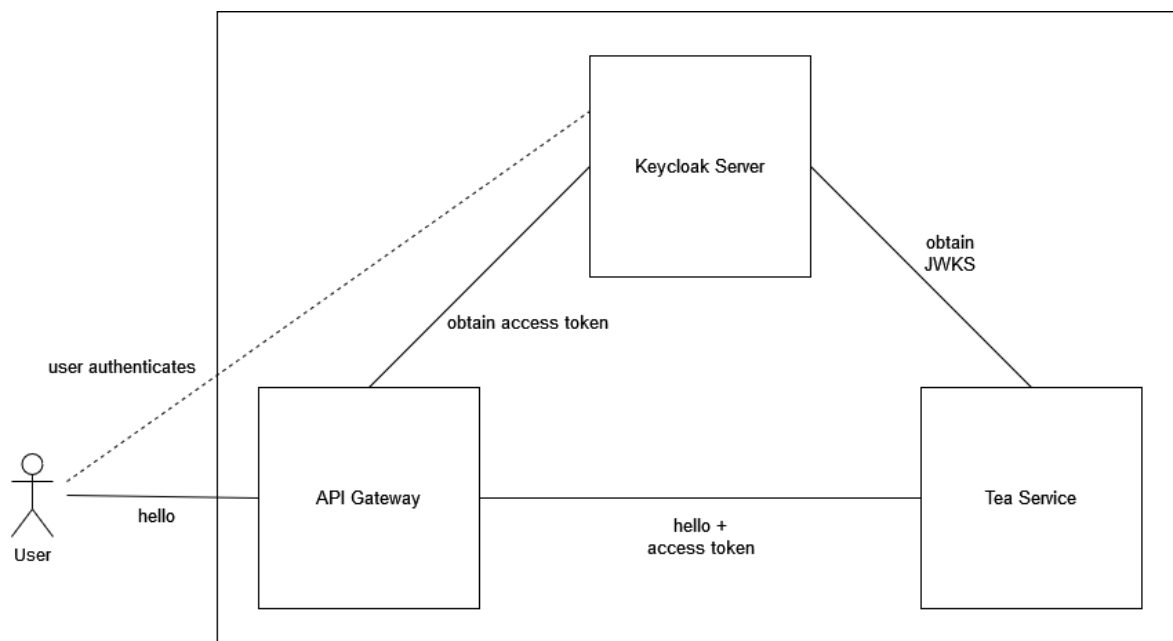


Figure 2.2: High level diagram of the implemented services and their relation to each other

frontend application to incorporate OAuth2 client functionality.

With this implementation, the first request to a protected resource, when the gateway hasn't obtained an access token yet, can be depicted as in figure 2.3

The detailed auth code grant flow has already been shown and explained in section ???, therefore a simplified version is depicted in this diagram.

Any subsequent request, as long as the access token is valid, is much simpler. The gateway already has an access token and all it has to do is append this access token to the routed request as authorization header and forward it to the Tea Service. This scenario is also used for load testing, as will be explained in section ???.

The second version does not implement a client at all. It is simply two resource servers in series. The gateway receives an access token with the request from the user, in theory via some frontend client, in the case it is sent by a jmeter script validates and forwards it to the Tea Service, which again validates the access token. In both versions, the Tea Service, or the Tea Service and the Gateway respectively must obtain the JSON Web Key Set (JWKS) from the Keycloak Server, so that they will be able to validate the access token. This happens at the first request.

The third version is again a copy of the other two but the Keycloak Server is not needed in this case and the services do not care about authorization at all.

2.2 Setup with Spring Boot and Keycloak

2.2.1 Spring and Spring Boot

!!! versionen All Services in this project were developed using Spring Boot¹ Version 3.0. Spring Boot is created on top of the Spring framework, a widely used open source application framework for Java. Spring provides dependency injection and different modules, like Spring Security, Spring Test or Spring ORM (object-relational mapping), among others [?]. Spring

¹<https://spring.io/projects/spring-boot>

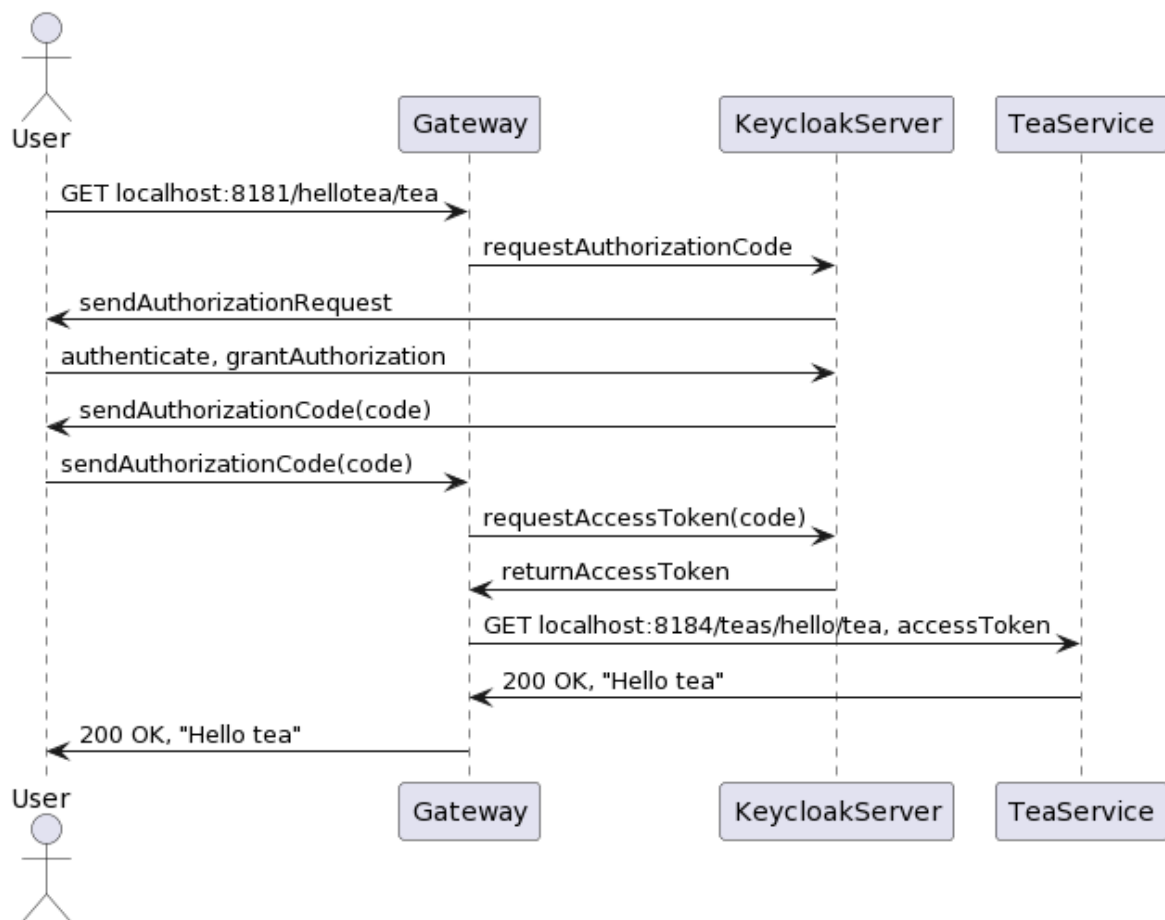


Figure 2.3: Sequence diagram of the first request to a protected resource including a simplified auth code grant flow

Boot was created in order to simplify the development of Spring-based applications by offering autoconfiguration and starter dependencies that bundle selections of libraries in one Maven or Gradle dependency [?, pp. 4f]. This helps to reduce the need for the developer to write boilerplate code manually, which means that one big advantage when using Spring boot is the quick project setup. However, these configurations can be overridden or customized when needed, like it is the case for security configurations [?, p. 50], either programmatically with Java or in many cases by adding configurations to the `applications.properties` or the `applications.yml` file [?]. For the Teapot project the `yml` variant was used whenever possible because this way configurations are easier to write and read, and therefore they are less error-prone.

Spring Boot projects can be initialized and downloaded with the Spring Initializr² which is also available when creating a new Project in IntelliJ. All Maven dependencies that are needed for a project can be chosen during project creation with Spring Initializr, or they can be added later to the `pom.xml` file.

2.2.2 Keycloak Server

Keycloak [?] is an identity and access management (IAM) platform. It is open source and published under the Apache Licence 2.0³. It supports OAuth2, OIDC and SAML. As an IAM, the Keycloak server can take care of user management, authentication of users and issuing access tokens and id tokens to registered clients. The version used for the Teapot project was 20.0 (Quarkus distribution). The Keycloak server was deployed locally in a docker container in dev mode. Keycloak supports different databases to store data, however, the default database (dev-file) was sufficient for the Teapot project. The Keycloak server already contains a `master realm` with the administrator account. The `master realm` is the parent of all other realms that can be created by an administrator. For this project, a `teapot realm` was created. Inside a realm, administrators can create (register) and manage clients and users. The Teapot Gateway needs to be registered as a client in the Teapot realm. When it is created, the client secret is set automatically for the new client, if `Client authentication` is enabled. This is possible because the Teapot Gateway is a confidential client. The secret is used by the Gateway application when connecting to the Keycloak server to authenticate itself.

For the Teapot project setup where all services are deployed with Docker Compose, the `Frontend-Url` is set to `http://host.docker.internal:10001` where the host name is the docker network and the port is the port assigned to Keycloak. If the `Frontend-Url` is not set explicitly, the host name for the Keycloak endpoints that are used for authentication and authorization flows is set to `localhost`. Services in other Docker containers access the Keycloak server under its `host.docker.internal` url. The `frontend-url` also determines how the `iss` claim is set in access tokens, which must be identical with the `issuer-uri` set at the resource server(!!! Quelle). If `iss` claim and `issuer-uri` do not match, the access token does not pass the validation and a 401 response will be sent back with a remark in the `XXX-Authenticate` header that the `iss` claim is not valid (see figure 2.4).

All endpoints of the Keycloak server for a realm are accessible under `<host:port>/realms/<realmname>`. OAuth2 resource servers and clients with the correct `issuer-uri` can call this endpoint to retrieve the other necessary endpoints, like `authorization_endpoint`, `token_endpoint`, `jwks_uri`, etc., but also other necessary information like supported signing algorithms, grant types, etc. Figure 2.5 shows the first part of these endpoints.

²<https://start.spring.io/>

³<http://www.apache.org/licenses/LICENSE-2.0>

2 Implementation

▼ Response Headers

```
WWW-Authenticate: "Bearer error='invalid_token', error_description='An error occurred while attempting to decode the Jwt: The iss claim is not valid', error_uri='https://tools.ietf.org/html/rfc6750#section-3.1'"
```

Figure 2.4: Response header indicating that the iss claim is not valid.

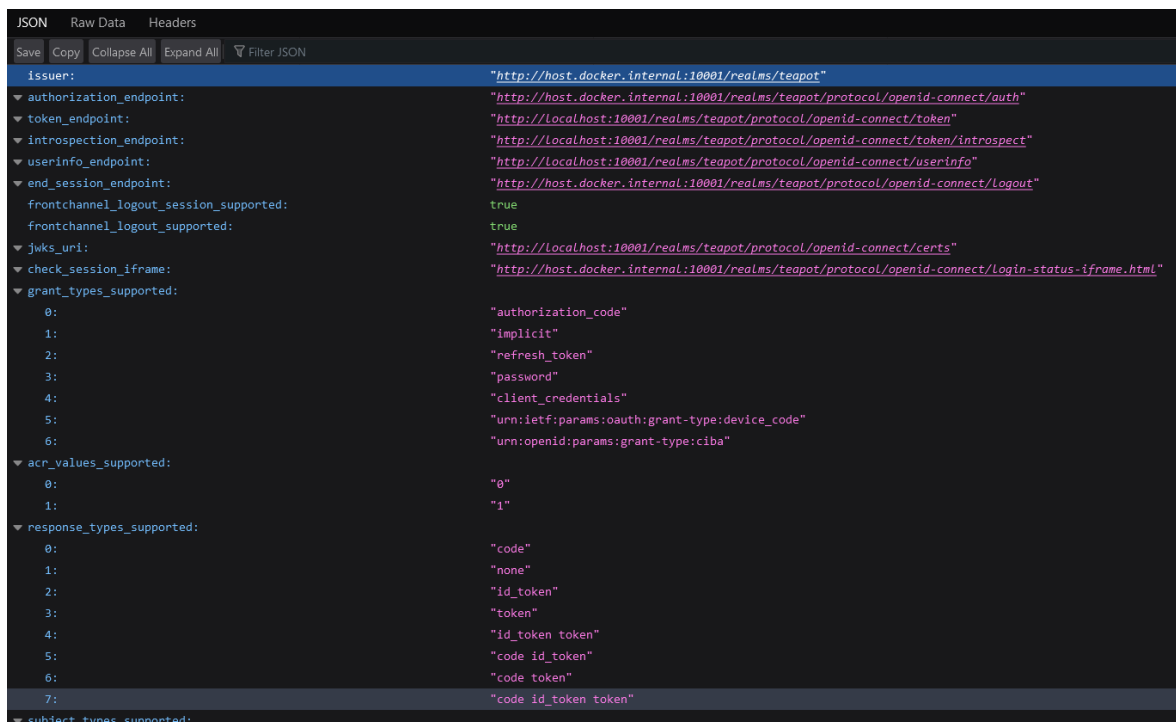


Figure 2.5: Keycloak endpoint configuration for the teapot realm (not complete).

2.2.3 Spring Cloud Gateway as OIDC Client

The Gateway's job in a MSA is to route requests to services beyond. There is a special Spring Boot starter dependency, `spring-cloud-starter-gateway`, that was used for the implementation of the Teapot project. Maven dependencies are injected in the `pom.xml` file in the following way:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

With the Spring Cloud Gateway implemented, a Handler Mapping checks incoming requests for matches with configured routes and if so, forwards them to the Gateway Web Handler. The request then goes through a filterchain where route-specific pre- and post-logic is applied[?].

Routes can be configured in the `application.properties` file or in the `application.yml`. Figure 2.6 shows an example route configuration from the `application.yml` file in the reduced Teapot project where no discovery service is used. The `uri` value is given as an environment variable and will be injected via the `docker compose.yml` file (!!! see docker). With the Eureka discovery service in the first Teapot, the value would be `lb://` followed by the name that the Tea service application uses to register with the discovery service. This way the Gateway does not have any need to know the specific current address of the Tea Service or any other application it is routing a request to. The Path predicate defines the path for the endpoint at the gateway. So in this case, requests to `http://localhost:8181/hellotea/U1a` will be recognized as a match for `$TEAS/teas/hello/U1a`, the path that is set under filters with the `SetPath`. `U1a` is an example value for the name variable.



```
1  server:
2    port: 8181
3
4  spring:
5    application:
6      name: gateway2
7    cloud:
8      gateway:
9        routes:
10       - id: helloTea
11         uri: ${TEAS}
12         predicates:
13           - Path=/hellotea/{name}
14         filters:
15           - SetPath=/teas/hello/{name}
```

Figure 2.6: Example route configuration from the Gateway's `application.yml` file in the reduced Teapot project

In order to configure the Gateway as OAuth2 client, we also need to include the `spring-boot-starter-oauth2-client` dependency in the `pom.xml` file:

2 Implementation

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

Here it is important to choose the correct starter dependency and to not get confused by the different oauth2-client dependencies available, as there are many with similar names. The `spring-boot-starter-oauth2-client` dependency is intended to be used with Spring Boot [?]. Then, after having created the client in Keycloak (see section 2.2.2), the application needs to be configured so it can connect to the authorization server and register with the client's credentials. All this is done in the `application.yml` file (see listing 2.1).

```
1 spring:
2   [...]
3   security:
4     oauth2:
5       client:
6         provider:
7           keycloak-provider:
8             issuer-uri: ${keycloak.server-url}/realms/teapot
9         registration:
10          keycloak-gateway-client:
11            provider: keycloak-provider
12            scope: openid
13            client-id: teapot-gateway
14            client-secret: ${client-secret}
15            authorization-grant-type: authorization_code
16            redirect-uri: 'http://localhost:8080/login/oauth2/code/{
  registrationId}'
```

Listing 2.1: OAuth2 client configuration in the Gateway's `application.yml` file

For this purpose we use the `spring.security.oauth2.client.registration` base property prefix, followed by the registration id that will be used by Spring Security's `OAuth2ClientProperties` class. In this project the client's registration id is `keycloak-gateway-client`. As explained in section ??, `openid` must be included in the scope claim. Further, the `client-id` and the `client-secret`, as well as the `authorization-grant-type` and the `redirect-uri` are specified. The `redirect-uri` is the address that the authorization server will send to the user agent to redirect the user back to the application after authorization has been granted (see section ??). The provider section contains the provider name, in this case `keycloak-provider`. This is the name which the registration section refers to. The `issuer-uri` must be set correctly, otherwise the application won't be able to start successfully. This also happens when the OIDC provider is not reachable. The reason is, that the `issuer-uri` is used by the application to retrieve vital configuration metadata from the OIDC provider which is needed for the creation of automatic configuration. As a default, a OpenID provider Configuration Request is made to "[specified issuer-uri]/.well-known/openid-configuration". This endpoint offers all the necessary configuration metadata, like `token_endpoint`, `jwks_uri`, `end_session_endpoint`, supported grant types and response types, supported signing and encryption algorithms etc [?].

The Gateway must also be able to attach access tokens to any authorized request that will be routed to a downstream resource server. Spring Security offers a `TokenRelayGatewayFilterFactory`

2 Implementation

which fetches the access token from the authenticated user and attaches an `Authorization` header to the request with the value `"Bearer" + token`. The fastest way to add the `TokenRelayGatewayFilterFactory` is certainly to add a `default-filter` to the route configuration in the `application.yml` file as shown in listing 2.5. This filter will then be applied to all configured routes. Alternatively, the filter can be configured for specific routes by adding `- TokenRelay=` to filters [?].

```
1 spring:
2   application:
3     name: gateway2
4   cloud:
5     gateway:
6       routes:
7
8       [...]
9
10      - id: milk
11        uri: ${MILK}
12        predicates:
13          - Path=/milk
14        filters:
15          - SetPath=/getmilk
16
17      default-filters:
18        - TokenRelay=
```

Listing 2.2: Route configuration with token relay default filter in the Gateway's `application.yml` file

Security configuration for the gateway's endpoints can now be added in the way that is shown in listing 2.4, taken from the reduced Teapot Gateway2. Because `/hellogateway` and `/hellotea/noauth` should remain open for testing purposes, this is taken care for with `permitAll()` before configuring all remaining endpoints as open for authenticated users only, with `.authorizeExchange().anyExchange().authenticated()`. With `oauth2login()` the users will be authenticated so they can have access to the protected endpoints [?].

```
1 @Configuration
2 @EnableWebFluxSecurity
3 public class Gateway2SecurityConfiguration {
4   @Bean
5   public SecurityWebFilterChain springSecurityWebFilterChain(
6     ServerHttpSecurity http,
7     ServerLogoutSuccessHandler handler) {
8     .authorizeExchange()
9     .pathMatchers("/hellogateway", "/hellotea/noauth")
10    .permitAll()
11    .and()
12    .authorizeExchange()
13    .anyExchange()
14    .authenticated()
15    .and()
16    .oauth2Login()
17    .and()
18    .logout()
19    .logoutSuccessHandler(handler);
```

2 Implementation

```
20         return http.build();
21     }
```

Listing 2.3: SecurityWebFilterChain for configuration of the OAuth2 client’s behaviour. Code example from the reduced Teapot Gateway2

One particular aspect here is the `logoutSuccessHandler` call that gets a `ServerLogoutSuccessHandler` object as an argument. A separate bean, as shown in listing ??, has to be written in order to make this work properly. The `OidcClientInitiatedServerLogoutSuccessHandler`, which implements the `ServerLogoutSuccessHandler` interface, takes care of the logout process and calls the Keycloak Server's `end_session_endpoint` for this user [?], [?], [?]. This process is defined in OpenID Connect Session Management 1.0 as the *RP-Initiated Logout*, where RP stands for relying party [?]. Because Keycloak provides Session Management and Discovery, the `end_session_endpoint` URL can be configured automatically with Spring Boot. The `postLogoutRedirectUri` is the URI that the user will be redirected to after having logged out successfully. User logout can be initiated by a GET or POST request to `base-url/logout` as default. The `/logout` endpoint does not need to be permitted explicitly in the filter chain [?]. Figures 2.7 and 2.8 show the process in the Firefox networks analytics tool. First, a POST request is sent to the Teapot Gateway's `logout` endpoint, then a redirect follows to `http://host.docker.internal:10001/realms/teapot/protocol/openid-connect/logout` which is the `end_session_endpoint` at the Keycloak, together with the `id_token_hint` and the `post_logout_redirect_uri` as query parameters. The `id_token_hint` is used to let Keycloak know for which user the session should be cancelled. The `post_logout_redirect_uri` is open to anonymous users and doesn't require authorization.

```

1  @Bean
2  public ServerLogoutSuccessHandler keycloakLogoutSuccessHandler(
ReactiveClientRegistrationRepository repository) {
3      OidcClientInitiatedServerLogoutSuccessHandler
oidcClientInitiatedServerLogoutSuccessHandler = new
OidcClientInitiatedServerLogoutSuccessHandler(repository);
4      oidcClientInitiatedServerLogoutSuccessHandler.
setPostLogoutRedirectUri("https://orf.at");
5      return oidcClientInitiatedServerLogoutSuccessHandler;
6  }

```

Listing 2.4: Logout success handler. Code example from the Teapot Gateway according to [?]


302	POST	localhost:8181	logout	document	html
302	GET	 host.docker.internal:10001	logout?id_token_hint=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXLTUxMjM0NTY3ODk5In0=	document	html

Figure 2.7: POST request to the
logout endpoint of the Teapot Gateway and redirection to Keycloak's
end_session_endpoint

With Spring Boot, a `GatewayApplication.java` class is created automatically, that contains the main method. With this setup the Gateway application is already fully functional and able to route requests to a resource server together with an access token after the user has authenticated successfully.

An additional feature in the Teapot Gateway is the `/hellogateway` endpoint which returns a string with a greeting to the user after reading the user's name from the authenti-

[illegible]

Figure 2.8: The end session endpoint with query parameters

cation principal . This is possible without adding an additional dependency because Spring Cloud Gateway already contains the spring-boot-starter-webflux dependency.

```
1 @RestController
2 public class GatewayController {
3     @GetMapping("/hellogateway")
4     public String greet(@AuthenticationPrincipal OAuth2User principal) {
5         return "Hello, " + principal.getName() + ", from Gateway";
6     }
7 }
```

Listing 2.5: Reading the user’s name from the authentication principal.

2.2.4 The Resource Server

The OAuth2 Resource Server receives and validates the access token and, if the token is valid, grants access to the requested resource (see section ??). The basic steps to configure a resource server with Spring Boot are not very different from the configuration of the OAuth2 client: implementation of the necessary dependencies in the pom.xml file, configuration of the issuer-uri, or optionally the jwk-set-uri in application.properties or application.properties and overriding the default SecurityFilterChain with a customized one [?].

The minimal dependencies needed are `spring-security-oauth2-resource-server`, which contains the resource server support, and `spring-security-oauth2-jose`, which allows the resource server to decode JWTs, and is therefore crucial for the application's ability to validate JWT access tokens [?]. Both are included in the `spring-boot-starter-oauth2-resource-starter` dependency. OAuth2 bearer token authentication is possible with JWTs or with opaque tokens (see section ??). The Teapot project works with JWT.

The authorization process when a request for a protected resource comes in without an access token, goes like this [?]:

- An unauthenticated request comes in from the User
- The `AuthorizationFilter` throws an `AccessDeniedException`
- The `ExceptionTranslationFilter` initiates *Start Authentication* and activates the `BearerTokenAuthenticationEntryPoint` to send a `WWW-Authenticate: Bearer` header (see figure 2.10)
- Now the client can retry the request with the bearer token.

When the request comes with a bearer token, the `BearerTokenAuthenticationFilter` extracts the token from the `HttpServletRequest` and creates a `BearerTokenAuthenticationToken`.

```
public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException) {
    HttpStatus status = HttpStatus.UNAUTHORIZED;
    Map<String, String> parameters = new LinkedHashMap();
    if (this.realmName != null) {...}

    if (authException instanceof OAuth2AuthenticationException) {...}

    String wwwAuthenticate = computeWWWAuthenticateHeaderValue(parameters);
    response.addHeader("WWW-Authenticate", wwwAuthenticate);
    response.setStatus(status.value());
}

no usages
public void setRealmName(String realmName) { this.realmName = realmName; }

1 usage
private static String computeWWWAuthenticateHeaderValue(Map<String, String> parameters) {
    StringBuilder wwwAuthenticate = new StringBuilder();
    wwwAuthenticate.append("Bearer");
    if (!parameters.isEmpty()) {...}

    return wwwAuthenticate.toString();
}
```

Figure 2.9: The `BearerTokenAuthenticationEntryPoint` sends a `WWW-Authenticate : Bearer` back to the requesting client [?]

GET http://localhost:8184/teas/getall

Params Authorization Headers (9) Body Pre-request Script Tests Settings

Body Cookies (1) Headers (12) Test Results

Key	Value
Set-Cookie	JSESSIONID=F
WWW-Authenticate	Bearer

Figure 2.10: `WWW-Authenticate` header in the response to an unauthorized request to the resource server

which implements the `Authentication` interface [?]. The `Authentication` represents the authenticated user and contains (among others) a principal, which represents an individual, corporation, login id or any other entity [?], credentials and authorities [?]. An authority is an instance of `GrantedAuthority` and usually represents coarse-grained permission, for example role or scope [?] (for more details about role-mapping, see section 2.2.5). The credentials in this case contain the access token. The [?] is then passed to the `AuthenticationManager`. The `AuthenticationManager` is selected by the `AuthenticationManagerResolver` based on the `HttpServletRequest`, either for JWT, like in this case, or for opaque tokens. The `AuthenticationManager` authenticates the `BearerTokenAuthenticationToken` which means in this case, that it validates the token, stored under `credentials`. Depending on wheather authentication fails (the token is not valid) or is successfull, the `SecurityContextHolder` is cleared out and the `AuthenticationEntryPoint` will send the `WWW-Authenticate` header again, or the `SecurityContextHolder` is set the with the `Authentication` object and the `FilterChain` continues [?].

Like with the OAuth2 client, the resource server needs the `issuer-uri` to be configured correctly. At startup the resource server application has to deduce the authorization server's configuration endpoint. With only the `issuer-uri` given, it is important that one of a set of specific configuration endpoints is supported. With the Keycloak server, the configuration endpoint is `http://localhost:10001/realms/teapot/.well-known/openid-configuration`. This endpoint can now be queried for the `jwks-url` property and for supported algorithms. With this information, the application can configure the validation strategy which will in the next step query the `jwks-url` for the public key set of these algorithms. Lastly, the validation strategy will be configured to check the `iss` claim of recieved JWT access tokens against the given `issuer-uri`. For this reason the authorization server must be up and reachable, otherwise the resource server application will fail at startup [?].

In order to allow the application to start independently when the authorization server is not yet reachable, the `jwk-set-uri` can be configured explicitly, because it doesn't need to call the `issuer-uri` in order to find out the end point to retrieve the JWKS [?]. In the Teapot project's `application.properties` file, this looks like in listing 2.6. Still, with the `issuer-uri` provided, the `iss` claim in incoming JWTs will be validated against the given issuer [?].

When a request is sent to a protected endpoint at the resource server, it uses the public key from the authorization server to validate the signature and match it with the token. Then the `exp` and `iss` claims in the token are checked [?]. For a more differentiated authorization policy it is possible to define scope and roles in Keycloak. In order to use them, custom claims have to be converted into authorities at the resource server (see section 2.2.5).

```

1 spring:
2
3 [...]
4
5   security:
6     oauth2:
7       resource-server:
8         jwt:
9           issuer-uri: ${KEYCLOAK}/realms/teapot
10          jwk-set-uri: ${KEYCLOAK}/realms/teapot/protocol/openid-connect/certs

```

Listing 2.6: `issuer-uri` and `jwk-set-uri` in the Tea service's `application.properties` file

Like with the client (section section 2.2.3 the `SecurityFilterChain` bean can be over-

2 Implementation

ridden for custom configuration. A minimal configuration of the `SecurityFilterChain` can look like in listing 2.7. The two endpoints `/teas/hello/noauth` and `/teas/create` are open for convenience during experimental development, while all other endpoints are protected and can only be accessed with a valid access token. `oauth2ResourceServer` gets a `Customizer` parameter of type `OAuth2ResourceServerConfigurer`. With this customizer it is possible to specify that JWT bearer tokens should be supported. This will populate the beforementioned `BearerTokenAuthenticationFilter` which is responsible for processing the access token [?].

```
1 @Configuration
2 public class TeaSecurityConfiguration {
3     @Bean
4     public SecurityFilterChain filterChain(HttpSecurity http) throws
5         Exception {
6         http
7             .authorizeHttpRequests().requestMatchers("/teas/hello/
8             noauth", "/teas/create").permitAll()
9             .anyRequest().authenticated();
10        http.oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);
11        [...]
12        return http.build();
13    }
14 }
```

Listing 2.7: `SecurityFilterChain` configuration in the Tea service (resource server)

For the definition of authorities that are not provided in the original access token, the customized conversion from the JWT to an `Authentication` object can be supplied at this point as `OAuth2ResourceServerConfigurer.JwtConfigurer.jwtAuthenticationConverter` instead of `OAuth2ResourceServerConfigurer::jwt` [?].

Spring Security has CSRF protection enabled by default [?]. Because the resource server relies on bearer token authentication, some authors in grey literature recommend making the session stateless [?] and as a consequence to disable CSRF protection [?]. This is only possible for resource servers, while clients that are consumed by browsers must always enable CSRF protection because they rely on session cookies [?]. This can be configured in the `SecurityFilterChain` bean as shown in listing 2.8.

```
1 @Bean
2 public SecurityFilterChain filterChain(HttpSecurity http) throws
3     Exception {
4     [...]
5     http.sessionManagement((session) -> session.sessionCreationPolicy
6     (SessionCreationPolicy.STATELESS))
7     .csrf().disable();
8     return http.build();
9 }
```

Listing 2.8: Stateless session and disabled CSRF protection configured in the `SecurityFilterChain` bean in the Tea service (resource server)

The actual resources that the Tea service is holding are Tea objects that are stored in a MongoDB database. However, the implementation of object relational mapping with spring boot lies beyond the scope of this thesis, therefore further details are omitted.

The gateway in a MSA can also be implemented as a resource server instead of being a client. In this case it is necessary that any client sending a request to the gateway is able to

provide the appropriate access token because it will not take care of redirecting the user to the OP for authentication. This version has been implemented for the purpose of answering the second research question of this thesis about the positioning of the oauth2 client. The steps to configure the gateway as a client are mostly covered already in this section and in section 2.2.3 and will therefore be omitted at this point.

2.2.5 Role mapping with Keycloak and Spring Boot Resource Server

In the event that an intruder manages to steal an access token, or if a member of the organization tries to perform operations that they are not entitled to, the possible damage can be significantly reduced by defining roles with different privileges. In the example of the Teapot this means that someone with a basic user role is not authorized to perform any operation that is not safe, like creating or deleting tea in the database. With a user token, only safe GET requests will be authorized by the resource server. Although this does not mean that no harm can be done by accessing information without altering it, the potential damage is limited compared to a misused access token for Teapot admins, who have privileges that enable them to erase the entire content of the Tea database. Of course this example is highly simplified and in real applications, even a basic user of the system might need to perform operations on the resource that would alter it, or have access to information not intended for anyone else, but the principle remains the same.

Role mapping therefore allows to apply the principle of least privilege (PoLP), which means that users only have the privileges necessary to perform what is inherent to their role, but nothing else (!!! Quelle). Figure 2.12 depicts the roles for the Teapot Gateway client, with their privileges described as defined in the Tea resource server (see listing 2.2.5).

Realm roles

Realm roles are the roles that you define for use in the current realm. [Learn more](#)

<input type="text" value="Search role by name"/> <input type="button" value="→"/> <input type="button" value="Create role"/>		
Role name	Composite	Description
default-roles-teapot ⓘ	True	<code>\$_role_default-roles</code>
offline_access	False	<code>\$_role_offline-access</code>
tea_admin	True	–
tea_user	False	–
uma_authorization	False	<code>\$_role_uma_authorization</code>

Figure 2.11: Roles in the Teapot realm with the custom `tea_admin` and `tea_user` roles.

With Keycloak there is the option to define realm roles and/or client roles. Realm roles can be composite by being associated to other realm roles or client roles [?]. Figure 2.12 shows the defined client roles for the Teapot Gateway, while figure 2.13 shows one of the composite realm roles, `tea_admin` and its associated client roles, `admin`, `textttuser` and `privileged_user` which belong to the `teapot-gateway` client.

Keycloak includes these roles in the access token in a `realm_access` or `resource_access` claim respectively. The access token for a user with the `tea_admin` realm role and the `admin`

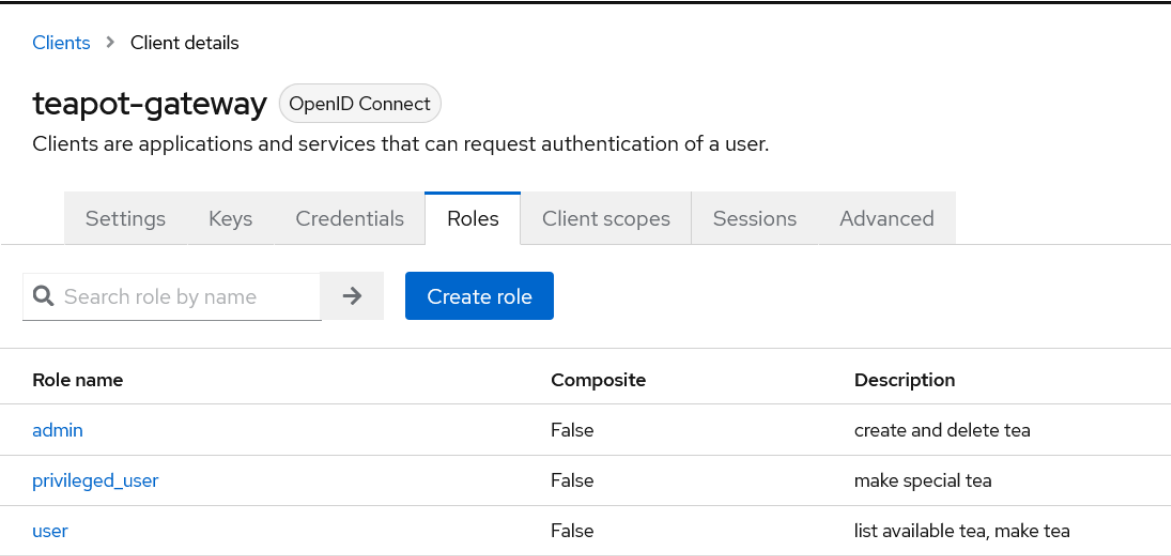


Figure 2.12: Example of client roles defined in the Keycloak admin console

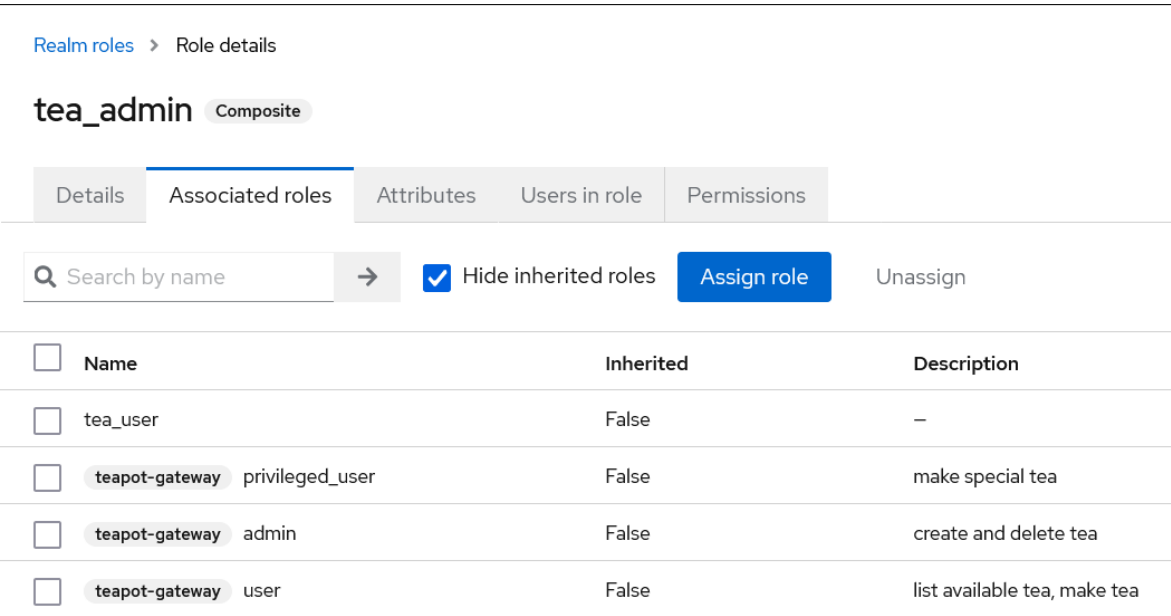


Figure 2.13: Example of a composite realm role and it's associated client role in the Keycloak admin console

2 Implementation

client role is shown in figure 2.14. The `realm_access` contains only the `tea_admin` role (the realm role) and `resource_access` contains all associated roles for the `teapot-client`.

[illegible]

Figure 2.14: Example of an access token with claims for realm and client roles, issued by the Keycloak server

As we can see, the access token does not contain authority claims. The `realm_access` or `resource_access` access claims are specific to Keycloak (!!! Quelle). Spring Security allows to check for authorities inside the authentication object, but it provides no means by default to check specifically for the access claims provided by a Keycloak access token. This means that these roles have no effect on the authorization process, unless an authority converter is added. A converter translates specific claims in the access token to an authority in order to distinguish roles in authorization. Listing ?? shows how the roles can be extracted from the specific claims in the access token. They are returned as a list of authorities and can be checked in the `SecurityFilterChain` by calling the `hasAuthority()` method. Spring Security also offers the `hasRole()` method, which checks for roles specifically. Roles are defined in Spring Security by the `ROLE_` prefix !!! Quelle!. This prefix has

to be added in the conversion process as well, as shown in listing ??, line 20. Spring Security provides a default `JwtAuthenticationConverter` for creating a `Authentication` from a JWT. This converter can be replaced by any class implementing `Converter<Jwt, AbstractAuthenticationToken>` [?]. Listing 2.10 shows the custom converter that is used in the Tea resource server. It returns a `JwtAuthenticationToken` which inherits from `AbstractOAuth2TokenAuthenticationToken` and contains the extracted realm roles and client roles from the access token as authorities. As it turned out, with the converter code proposed by [?] it is important that the profile scope is included in the access token, so that Keycloak will automatically also add further profile information, including the `preferred_username` path to the token. As an alternative the try-catch block was added, so that the `Authentication` is created only with the value of the sub claim.

```

1 @RequiredArgsConstructor
2 class JwtGrantedAuthoritiesConverter implements Converter<Jwt, Collection
  <? extends GrantedAuthority>> {
3
4     @Override
5     @SuppressWarnings({"rawtypes", "unchecked"})
6     public Collection<? extends GrantedAuthority> convert(Jwt jwt) {
7         return Stream.of("$.realm_access.roles", "$.resource_access.*.
  roles").flatMap(claimPaths -> {
8             Object claim;
9             try {
10                 claim = JsonPath.read(jwt.getClaims(), claimPaths
11             );
12             } catch (PathNotFoundException e) {
13                 return Stream.empty();
14             }
15             final var firstItem = ((Collection) claim).iterator()
16             .next();
17             if (firstItem instanceof String) {
18                 return (Stream<String>) ((Collection) claim).
19                 stream();
20             }
21             if (Collection.class.isAssignableFrom(firstItem.
22             getClass())) {
23                 return (Stream<String>) ((Collection) claim).
24                 stream().flatMap(item -> ((Collection) item).stream()).map(String.
25                 class::cast);
26             }
27             return Stream.empty();
28         })
29         .map(authority -> new SimpleGrantedAuthority("ROLE_" +
30         authority))
31         .map(GrantedAuthority.class::cast).toList();
32     }
33 }

```

Listing 2.9: Extraction of client roles and realm roles from Keycloak access token and conversion to granted authorities according to [?]. Simplified and with addition of the `ROLE_` prefix.

```

1 @Component
2 @RequiredArgsConstructor

```

2 Implementation

```
3 class SpringAddonsJwtAuthenticationConverter implements Converter<Jwt,
  JwtAuthenticationToken> {
4
5     @Override
6     public JwtAuthenticationToken convert(Jwt jwt) {
7         final var authorities = new JwtGrantedAuthoritiesConverter().
8         convert(jwt);
9         final String username;
10        try {
11            username = JsonPath.read(jwt.getClaims(), "preferred_username
12            ");
13            return new JwtAuthenticationToken(jwt, authorities, username)
14            ;
15        } catch (PathNotFoundException e) {
16            return new JwtAuthenticationToken(jwt, authorities);
17        }
18    }
19 }
```

Listing 2.10: Custom converter to set the extracted granted authorities from the access token in the new Authentication according to [?] with added try/catch blocks

The `SecurityFilterChain` bean can now be overridden in a way that access to different endpoints of the service is granted or not, depending on the role authority extracted from the access token of the user that requests a resource. The custom jwt converter from listing 2.10 is given as parameter to the `oauth2resourceServer` overload method. In listing 2.2.5, the `/teas/admin` endpoint is open for realm admins, while the `/teas/create` and `teas/delete/*` endpoints are open for client admins specifically. A user with the realm role `tea_admin` has therefore access to all protected endpoints because `tea_admin` is a composite role that also contains the three client roles. A user with only the client admin role assigned would not have access to `/teas/admin`, but can still access other endpoints. Because the client admin is not a composite role, the correspondent authority must be allowed explicitly in addition to the respective user roles (see lines 21-25 in figure 2.2.5).

```
1 @RequiredArgsConstructor
2 @Configuration
3 @EnableWebSecurity
4 public class TeaSecurityConfiguration {
5     public static final String REALM_ADMIN = "tea_admin";
6     public static final String CLIENT_ADMIN = "admin";
7     public static final String REALM_USER = "tea_user";
8     public static final String CLIENT_USER = "user";
9     public static final String CLIENT_PRIVILEGED_USER = "privileged_user"
10    ;
11
12    @Bean
13    public SecurityFilterChain filterChain(HttpSecurity http, Converter<
14    Jwt, ? extends AbstractAuthenticationToken> jwtAuthenticationConverter
15    ) throws Exception {
16        http
17            .authorizeHttpRequests().requestMatchers("/teas/hello/
18            noauth")
19            .permitAll()
20            .requestMatchers("/teas/admin")
21            .hasRole(REALM_ADMIN)
22            .requestMatchers("/teas/create", "/teas/delete/*")
23            .hasRole(CLIENT_ADMIN)
24            .and()
25            .oauth2ResourceServer().jwt().jwtAuthenticationConverter(jwtAuthenticationConverter)
26        ;
27    }
28 }
```

2 Implementation

```
19         .hasRole(CLIENT_ADMIN)
20         .requestMatchers("/teas/maketea/special")
21         .hasAnyRole(CLIENT_PRIVILEGED_USER, CLIENT_ADMIN)
22         .requestMatchers("/teas/maketea/*", "/teas/hello/user")
23         .hasAnyRole(CLIENT_USER, CLIENT_PRIVILEGED_USER,
CLIENT_ADMIN)
24         .requestMatchers("/teas/getall")
25         .hasAnyRole(REALM_USER, REALM_ADMIN, CLIENT_ADMIN)
26         .anyRequest().authenticated();
27     http.oauth2ResourceServer().jwt().jwtAuthenticationConverter(
jwtAuthenticationConverter);
28     http.sessionManagement().sessionCreationPolicy(
SessionCreationPolicy.STATELESS);
29     http.csrf().disable();
30     return http.build();
31 }
32 }
```

2.3 Load testing with JMeter

Schaun, wie floren und der andere mit bff das beschrieben haben

2.4 Code Analysis

LoC -> Dependencies? + Sonarqube Ergebnisse?

3 Discussion and Results

implementation: schwierig wegen deprecation (spring boot3, spring security, keycloak adapter) und nicht mehr aktuellen tutorials und documentations -> keycloak war am anfang noch am übergang und doc nicht up to date

access token issued by keycloak does not conform to the specifications for jwt access tokens in rfc 9068 by default. can be changed? not easy to find...

typ header parameter has JWT as value instead of at+jwt or application/at+jwt as required in RFC 9068 [?]. The RFC specifies further that resource servers must reject tokens containing any other value. This is not the case with the default Spring Security Configuration.

aud claim is missing "If the request does not include a "resource" parameter, the authorization server use a default resource indicator in the "aud" claim" bertocciJSONWebToken2021

Token validation with Spring Security can be configured with the OAuth2TokenValidator API.

userinfo endpoint returns only the sub claim. keycloak documentation refers to oidc specification. config möglich über client scopes -> scope anklicken -> mappers -> add to id token, add to access token, add to userinfo... + assign to role: client scopes -> scope anklicken -> scope -> assign to role ist ein bissl verwirrend und keine doku zu finden.

scaling: api gateway has the potential to become a bottleneck [?] -> bff can mitigate this to a certain degree by desing by having several gateways, but also a basic API gateway should be duplicated at some point.

3.1 Response times

JMeter Results

!!! Wenn alles nach best practice implementiert ist, könnte es sein, dass die zeiten nicht mehr in der reihenfolge liegen scaling... welche art von authentication zwischen gateway als rs und tea rs, könnte auch mtls sein z.B.

4 Conclusion and Future Work

future work: - best practice, not only poc, e.g. better client authentication (signed jwt...), mTLS, etc. - testing

oauth 2.1: "Including the client credentials in the request content using the two parameters is NOT RECOMMENDED and SHOULD be limited to clients unable to directly utilize the HTTP Basic authentication scheme (or other password-based HTTP authentication schemes)" -> ob das bei mir passiert, seh ich in wireshark. wenn ja, sind die ganzen tutorials schuld.

List of Figures

1.1	A very abstract illustration of a monolith and microservices according to [?]	1
1.2	Steps of the authorization code grant in the interaction between all roles as described in [?]	5
1.3	Token response from the Keycloak server in Postman	6
2.1	High level diagram of the implemented services and their relation to each other	10
2.2	High level diagram of the implemented services and their relation to each other	11
2.3	Sequence diagram of the first request to a protected resource including a simplified auth code grant flow	12
2.4	Response header indicating that the iss claim is not valid.	14
2.5	Keycloak endpoint configuration for the teapot realm (not complete).	14
2.6	Example route configuration from the Gateway's application.yml file in the reduced Teapot project	15
2.7	POST request to the logout endpoint of the Teapot Gateway and redirection to Keycloak's end_session_endpoint	1
2.8	The end_session_endpoint with query parameters	19
2.9	The BearerTokenAuthenticationEntryPoint sends a WWW-Authenticate : Bearer back to the requesting client [?]	20
2.10	WWW-Authenticate header in the response to an unauthorized request to the resource server	20
2.11	Roles in the Teapot realm with the custom tea_admin and tea_user roles.	23
2.12	Example of client roles defined in the Keycloak admin console	24
2.13	Example of a composite realm role and it's associated client role in the Key- cloak admin console	24
2.14	Example of an access token with claims for realm and client roles, issued by the Keycloak server	25

Listings

2.1	OAuth2 client configuration in the Gateway's application.yml file	16
2.2	Route configuration with token relay default filter in the Gateway's application.yml file	17
2.3	SecurityWebFilterChain for configuration of the OAuth2 client's behaviour. Code example from the reduced Teapot Gateway2	17
2.4	Logout success handler. Code example from the Teapot Gateway according to [?]	18
2.5	Reading the user's name from the authentication principal.	19
2.6	issuer-uri and jwk-set-uri in the Tea service's application.properties file	21
2.7	SecurityFilterChain configuration in the Tea service (resource server) .	22
2.8	Stateless session and disabled CSRF protection configured in theSecurityFilterChain bean in the Tea service (resource server)	22
2.9	Extraction of client roles and realm roles from Keycloak access token and conversion to granted authorities according to [?]. Simplified and with addition of the ROLE_ prefix.	26
2.10	Custom converter to set the extracted granted authorities from the access token in the new Authentication according to [?] with added try/catch blocks	26

List of Tables