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

!!! 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: Wie lassen sich beide Patterns mit Spring Boot implementieren? Welche Unterschiede gibt es zwischen den Varianten, z.B. in den Bereichen Performance, Sicherheit, Komplexität?

1.1 Related Work

1.2 Microservice-based vs Monolithic Architecture

Was ist MSA

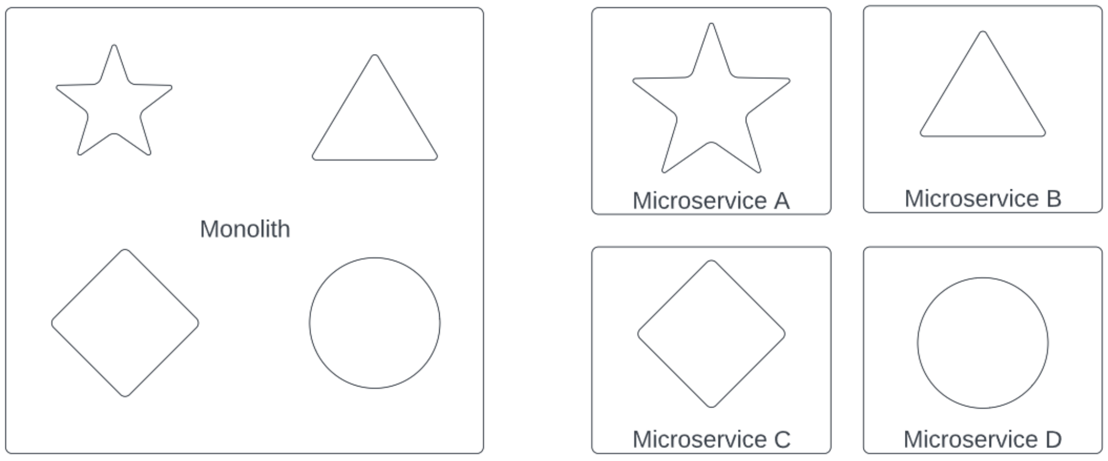


Figure 1.1: A very abstract illustration of a monolith and micoservices according to [**?**]

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 posess 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 OAuth2

Neu: redirect-uri! Wird beim gateway erwähnt!

OAuth 2.0, also often referred to as OAuth2, is an open protocol for delegated authorization, defined by the Internet Engineering Task Force (IETF) in the Request for Comments (RFC) 6749 [?] and RFC 6750 [?]. Authors of grey and academic literature seem to agree that OAuth2 is the standard for authorization in MSA environment (see chapter ??). OAuth2 was developed to allow a third-party client to access a certain (protected) resource on behalf of the owner of this resource [?]. In a MSA environment, where different services and possibly an API Gateway have to communicate with each other on behalf of a user or of another service, this concept of third-party access makes OAuth2 a feasible solution. The access to a resource happens by means of a so-called *access token*, which is issued to the client from an authorization server and which allows the client, now in possession of this token, to access the protected resource. The token now has to be sent with every request to the server holding that resource. This chapter gives insight into some of the mechanisms and specifications of the OAuth 2.0 protocol. However, this thesis can not cover all the details of OAuth2 and some concepts have to be described in a simplified manner.

1.4.1 OAuth2 Roles

There are four important roles in the OAuth2 authorization flow [?]:

- The *resource owner* is the person or entity that owns a protected resource. The resource owner can grant access to this resource to a third party.
- The *resource server* is the server where the resource in question lives. It responds to requests containing the access token.
- The *client* is any application (e.g. a web application or a mobile application) requesting the resource. It is not specified where this application is executed. The client can also act on its own behalf when it is the resource owner at the same time.
- The *authorization server* is the server responsible for authentication of the resource owner, obtaining authorization and issuing access tokens to the client.

An example scenario to illustrate these roles would be that a user (the resource owner) has a Facebook account and wants another application (the client) to access data (the protected resource) in their account, maybe because the app has promised to analyze the user's personality based on their timeline posts [?].

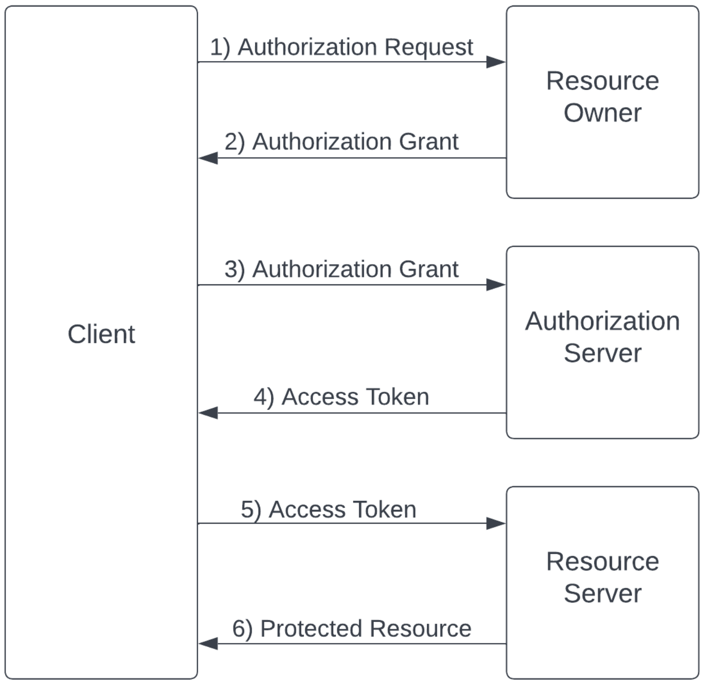


Figure 1.2: Abstraction of the OAuth2 flow after [?, fig. 1]

1.4.2 The OAuth2 Authorization Flow

The simple and most dangerous way for the client to access the user's Facebook data from the example above would be that the user passes their username and password to the client, which can then comfortably log in to the user's Facebook account and do with it whatever it wants to do [?, p. 81]. Obviously, this would lead to many problems if the client, now in possession of the user's credentials, is not trustworthy. OAuth2 solves this problem by enabling the user to grant the client access to their data without letting it see their username and password. This is done by delegating the authorization process to the authorization server. Once the user is authenticated with the authorization server and has given permission to the application to access data from their account, the application will receive the access token and can present this token to the server in exchange for the data they want. The basic steps of the OAuth2 flow are as follows [?] (see also figure 1.2):

1. An authorization request is made by the client to the resource owner (preferably via the authorization server)

2. An authorization grant is issued (again preferably via the authorization server) to the client.

3. The client requests an access token from the authorization server by presenting the authorization grant.

4. The access token is issued to the client (after authenticating the client and validating the request).

5. The client presents the access token to the resource server and requests the protected resource.

6. The access token is validated by the resource server (locally or by calling the authorization server) and, if successful, responds with the requested resource.

In reality the authorization grant, which represents the authorization by the resource owner for the client to access a resource, can come in different shapes.

1.4.3 The OAuth2 Authorization Grant Types

The exact flow in which the client can receive the access token can differ, depending on the *grant type.* The original specification defines four different grant types, but it is also possible to define additional grant types [?]. However, not all of those original grant types are still recommended for implementation according to the OAuth 2.0 Security Best Current Practice (BCP) IETF Internet Draft [?].

- With the *authorization code grant*, the authorization server responds to the authorization request (after authenticating the resource owner and obtaining authorization) not with the access token, but with a code, which the resource owner's user-agent (e.g. web-browser) will pass on to the client at the redirection URI [?]. The client can then exchange this code for an access token directly with the authorization server, without exposing the token to the resource owner or potentially anyone else. It is also possible for the authorization server to authenticate the client [?].

- The *client credentials* grant: The client receives an access token after authenticating itself to the authorization server with its client credentials (a password or public/private key) [?].

Legacy grant types [?]:

- The *implicit grant* is similar to the authorization code grant, but here the access token is sent to the client directly instead of sending a code first and the authorization server does not authenticate the client [?].

- With the *resource owner password grant*, `password grant` for short, the resource owner's credentials are directly exchanged for an access token [?].

Other grant types are the *device code* grant, which is an extension [?], and sometimes also the *refresh token* [?] is called a grant type [?], [?, p. 372], although it is not considered as such in the original OAuth2 specification [?].

The choice of the grant type depends strongly on the type of the client. The authorization code grant type is optimized for confidential clients, which are capable of keeping their client credentials secret [?]. Public clients, such as web-browser applications or native (mobile) applications, which cannot maintain confidentiality of their secrets were intended to use the implicit grant type in the original specification. The problem with this grant type is that the access token is not issued to the intended client directly, but is handed over by the user-agent, as in `client.example/redirection_endpoint#access_token=abcdef`, and URLs are often stored in browsing histories [?]. So in order to prevent access token leakage and replay attacks, it is now recommended that public clients should use the authorization code grant as well, with mandatory Proof-Key for Code Exchange (PKCE) [?]. The PKCE, pronounced "pixie", enables clients that are not able to maintain a secret to use the authorizaton grant flow, but it is recommended for all clients [?]. Also the client credentials grant is reserved for confidential clients only [?], and is used mostly for interaction between systems when no end-user is present, for example a web application accessing an API for metadata [?, p. 372]. In this case, access to the protected resource happens on the client's own behalf.

Although the discussion about whether or not to use certain grant types has gone on for some years already, the OAuth2 BCP, in which the use of the implicit grant is discouraged and the password grant type is dismissed altogether, is rather new: It was first mentioned in version 9 of the OAuth 2.0 BCP in 2018 that "Clients SHOULD NOT use the implicit grant" [?] and in version 11 that "The resource owner password credentials grant MUST NOT be used" [?]. Therefore, it is important to pay particular attention to the date and origin of tutorials and articles about OAuth2 in order to avoid receiving old information.

1.4.4 OAuth2 Tokens and Validation

By some aspects, the access token can be seen as a key to a door [?]. It does not contain any information about the person using it, but as long as the key fits, the door will open. But unlike a key, one important security feature with OAuth2 access tokens is that they can expire. To spare the user the effort of having to grant permission again each time the token expires, the client application gets a refresh token along with the access token and once the access token has reached its defined expiration time, the client can then call the authorization server and exchange the refresh token for a new access token. A short expiration time makes it possible to validate access tokens locally, thus reducing the number of necessary calls. In case a permission to a client (represented by the access token) gets revoked at the authorization server, the service will not know this and the access token appears valid with local validation, but only until it expires. Therefore, it is important to evaluate carefully where in the MSA local validation is sufficient and where validation with the authorization server is necessary for higher security [?]. In any case, it is highly recommended to never accept unvalidated access tokens [?].

The nature of the access token is not defined in the OAuth2 specification. It can be an arbitrary string that serves as a reference to the authorization information, or a self-contained token [?]. A reference token is susceptible to brute force attacks, therefore additional strategies to prevent brute forcing must be implemented [?, 121]. In order to validate a reference token, a call to the issuing authorization server is inevitable. On the other hand, a self-contained token can be validated locally by means of the signature it carries [?, p. 121]. A very popular format for access tokens is the JSON Web Token (JWT) format, which will be discussed in more detail in section ??. In any case, the access token is defined as a string that represents the authorization for the client, but also the scope and duration of access [?]. It has no meaning to the client, similar to how a person does not need to know how a lock and key work in order to open a door.

The definition for access token privilege restriction in RFC 9068 [?] states that access tokens should be restricted to a specific resource server (several resource servers are possible, but preferably only one). This prevents clients as well as users from exceeding their privileges. Resource servers at the other end must check if they are the intended resource server for that access token [?]. The scope is defined by the authorization server. It is not mandatory for a client to ask for a specific scope, but if it does not do that, the authorization server must either fail the request altogether, or issue an access token containing a default scope [?].

Next to the access token, there is also a *refresh token* [?]. It can be issued to the client together with the access token (with certain grant types) and permits the client to request a new access token when the previous one has expired [?]. With the refresh token it is possible to define short expiration times for access tokens. The new token will not be issued when the permission has been revoked, but as long as this is not the case, the new token can be minted without bothering the resource owner.

1.4.5 JSON Web Token (JWT)

The JSON Web Token format (JWT) is a compact format for transmitting information (also known as claims) between two parties over HTTP and it is defined by the IETF in RFC 7519 [?]. It has become a popular choice for the use as access token in OAuth2, as described in the JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens specification, RFC 9068 [?], because it is self-contained and gives the possibility to be validated locally by the resource server (within the limitations discussed in section ??). While it is possible to implement an authorization mechanism using JWTs without the OAuth2 protocol, this topic lies outside the scope of this thesis. The following section will focus on the nature of JWT in general and on its use as OAuth2 access token.

A JWT is a JSON object, encoded in a JSON Web Signature (JWS) or JSON Web Encryption (JWE) structure, or both [?]. This means, it offers the possibility to be cryptographically signed and/or encrypted. Although it is possible to transmit unsigned JWTs, signing JWT access tokens is now mandatory for OAuth2 [?]. Signature algorithms can be either symmetric or asymmetric, but for OAuth2 access tokens, it is recommended to use asymmetric cryptography, and RS256 must be supported by authorization servers as defined in RFC 9068 [?], but experts have recommended this for some time already, e.g. [?].

A signed JWT consists of three elements, each of them base64-encoded and separated with a ".". The first element is the JavaScript Object Signing and Encryption (JOSE) header, the second is the JWT payload and the third is the signature [?]. Typically, the JOSE header contains the `typ` parameter (defined in the JWT specification [?], which should have `JWT` as a value, and, more specifically for OAuth2 access tokens, it must be `at+jwt` [?]. Special attention will be paid also later in this thesis to the `alg` parameter, which is not defined in the JWT specification, but in the specification for JWS in RFC 7515 [?]. The `alg` parameter indicates the algorithm used to cryptographically sign the JWT and the respective value must be either registered in the IANA "JSON Web Signature and Encryption Algorithms" registry, or contain a collision-resistant name. As per the RFC 9068, it must never contain "none" as a value. Finally, the `kid` (key ID) parameter contains a hint about the key that was used to sign the JWS [?]. It is optional and can be used to indicate a key change.

The second element of the JWT is the JWT claims set [?], or JWT payload [?, p. 160]. It contains the "business data" [?, p. 160] of the JWT. The JWT specification does not define which claims are mandatory, but rather leaves this to the specific applications to define. However, it is defined that only claims that are understood by the recipient can be accepted. The JWT specification also defines a list of "registered Claim Names", which are not intended to be mandatory, but are intended as a starting point for further specification. Out of these, the following claims, all defined in RFC 7519 [?], are required for the use in JWT access tokens [?]:

- `iss`: The issuer of the JWT.
- `exp`: The expiration time, after which a token must not be processed any more.
- `aud`: This parameter identifies the resource for which the access token is intended. It is mandatory as per RFC 9068 in order to prevent cross-JWT confusion, so access tokens issued by the same authorization server for different resources remain unique [?].
- `sub`: The subject of the JWT, either the resource owner (authorization code grant) or the client (client credentials grant), depending on whether a resource owner is involved in granting access [?].
- `iat`: Issuing time of the token.

- `jti`: JWT ID, a unique identifier for the JWT.

The kind of access that is requested can be specified within the `scope` parameter in the request from the client to the authorization server, but often it also contains identifiers for the resource itself or its location [?]. To lessen the burden on the `scope` parameter, there is also a more recent RFC that defines `resource` [?] for the this purpose. However, in the access token, the resource server is indicated not by the `scope` parameter, but by the `aud` parameter [?]. Additionally, the `client_id` claim, as defined in the RFC 8693 [?], as the name suggests, identifies the OAuth2 client that requested the access token. When using OIDC, other optional claims may become relevant, such as `auth_time`, `acr` and `amr`, which are defined in the OpenID Connect Core specification [?]. When first-party clients invoke a backend API belonging to the same solution, it is common that resource owner attributes are carried in the access token.

The access token is issued in response to a request by the client, as in Listing 1.1. The token corresponding to the example request in listing 1.1 can be seen in listing 1.2.

```

   GET /as/authorization.oauth2?response_type=code
           &client_id=2349832dg8s7f87
           &state=123456789
           &scope=%read%write%delete
           &redirect_uri=https%3A%2F%2Fclient%2Eulala%2Enet%2Fcb
           &resource=https%3A%2F%2Frs.ulala.com%2F HTTP/1.1
        Host: authorization-server.ularauch.net
```

Listing 1.1: Example request for an access token according to [?]

```
Header:

      {"typ":"at+JWT","alg":"RS256","kid":"RjEwOwOA"}

   Claims:

{
    "iss": "https://authorization-server.ularauch.net/",
    "iat": "2022-12-31T19:02:23.942Z",
    "exp": "2022-12-31T19:12:23.942Z",
    "aud": "https://rs.ulala.com/",
    "sub": "5ba552d67",
    "jti": "dbe39bf3a3ba4238a513f51d6e1691c4",
    "client_id": "s6BhdRkqt3",
    "scope": "read write delete"
}
```

Listing 1.2: Example JWT access token according to [?]

1.4.6 OpenID Connect (OIDC)

!!! ALT

Today, when reading about OAuth2, the warning that OAuth2 should not be used for authentication is hard to overlook. Still, authentication is an important component in order to secure a system and OAuth2 can be used *within* an authentication scheme [?]. With OAuth2, the resource owner will authenticate to the authorization server and also the client has to authenticate to the authorization server in many cases, but it is not the concern of

OAuth2 *how* the authentication is done [?]. In this context it is useful to understand that for the client the access token has no meaning and will just be passed on to the resource server for validation. The client does not learn anything about the user and the fact that an access token was issued should not be misunderstood as a proof that the end-user was correctly authenticated [?]. When information about the user is needed, OAuth2 is therefore not sufficient to cover authentication, even if this has not been and might still not be an unusual practice [?], [?]. The problems and pitfalls associated with the use of OAuth2 for authentication purposes are discussed more in detail in [?]. Instead, OpenID Connect (OIDC) is a layer on top of the OAuth2 specification and has been developed exactly for this purpose.

OpenID Connect 1.0 (OIDC) is an open protocol defined as a layer on top of OAuth2 by the OpenID Foundation [?] in 2014. Often there is a need for clients to be able to identify end-users, and OAuth2 does not fulfil this purpose, because it is not intended to be used for authentication. OIDC was developed to close this gap [?].

An OIDC flow is very similar to the OAuth2 flow, with a small, but significant difference: in addition to the access token, the authorization server, which is also responsible for handling authentication of the end-user, thus now being an OIDC provider or authentication server, issues also an ID token [?], [?]. The client can also send the access token to the UserInfo Endpoint (at the OIDC provider), which will return a defined set of additional standard claims about the user [?]. The OIDC flow consists of the following steps [?]:

1. Authentication request from the client to the OIDC provider

2. Authentication of the end-user at the OIDC provider + obtaining authorization

3. ID token (and usually access token) issued by OIDC provider to client

4. UserInfo request with access token from client to UserInfo endpoint

5. UserInfo response from UserInfo endpoint to client

The OIDC specification provides three specific authentication flows [?]:

- The *authorization code flow*, similar to the process described for the authorization code grant in section 1.4.3, but an ID token is issued to the client together with the access token.

- The *implicit flow*, again similar to the OAuth2 implicit grant. The OIDC provider redirects the end-user to the client, together with the ID token and the access token.

- The *hybrid flow* combines characteristics from both other flows. Clients receive always an authorization code and additionally the access token or the ID token. The other token can be exchanged for the authorization code.

As per the OAuth2 specification, an access token is opaque to the client [?]. In order to maintain this requirement, the ID token carrying information for user authentication is a separate token, issued alongside the access token [?]. The ID token is a JWT, containing claims similar to the OAuth2 access token, such as `iss, aud, exp, iat` (see section **??**, but also the `sub` claim, to uniquely identify the subject (end-user) with the client, `nonce`, which is used to prevent replay attacks and to associate the ID token with a client session, and other optional claims (`acr, amr, azp`). Other claims are possible as well, however, claims must be understood or be ignored otherwise. An example for an ID token is given in listing 1.3, where also the `auth_time` claim is used, denoting the time when the user has authenticated. An OIDC authentication request is an OAuth2 authorization request where the scope parameter must be present with open_id as a value. Other values for open_id can be present as well [?].

```
{
  "iss": "https://server.ularauch.net",
  "sub": "24400320",
  "aud": "s6BhdRkqt3",
  "nonce": "n-0S6_WzA2Mj",
  "exp": 1311281970,
  "iat": 1311280970,
  "auth_time": 1311280969,
  "acr": "urn:mace:incommon:iap:silver"
 }
```

Listing 1.3: Example for an ID token according to [?]

```
 HTTP/1.1 200 OK
 Content-Type: application/json

 {
  "sub": "248289761001",
  "name": "Ula Rauch",
  "given_name": "Ursula",
  "family_name": "Rauch",
  "preferred_username": "ulala",
  "email": "ursula.rauch@stud.fh-campuswien.ac.at",
  "picture": "http://ularauch.net/ulala/ula.jpg"
 }
```

Listing 1.4: UserInfo Response example according to [?]

Although the ID token appears to be very similar to an access token, there are some important differences to be pointed out [?]:

- The audience: ID tokens should only be sent to and read by the OAuth2 client. Consequently, ID tokens should never be sent to an API. Access tokens should be read only by the API (the resource server) it was meant for, but never by the client.
- The format: the format for access tokens is not specified, it can be a JWT, but it can also be an arbitrary string, while on the other hand an ID token is always a JWT.

OIDC also defines a protected resource at the OIDC provider, the UserInfo endpoint, where the client can request a set of standard claims with meta-data about the user in question in exchange for the access token [?]. An example for these claims is shown in listing 1.4. Also, like in the initial authentication request, the scope parameter must be present with the value open_id in the request for userInfo claims [?].

1.5 The positioning of the OAuth2 client in a MSA system

!!! nocheinmal anschaun. 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 complextiy 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 Postman[1] and Wireshark[2].

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 JMeter[3]. Details about the setup are documented in section 2.3. (!!! code complexity und security?)

[1]https://www.postman.com/
[2]https://www.wireshark.org/
[3]https://jmeter.apache.org/

2 Implementation

blabla
hier die Hardware specifications?

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 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

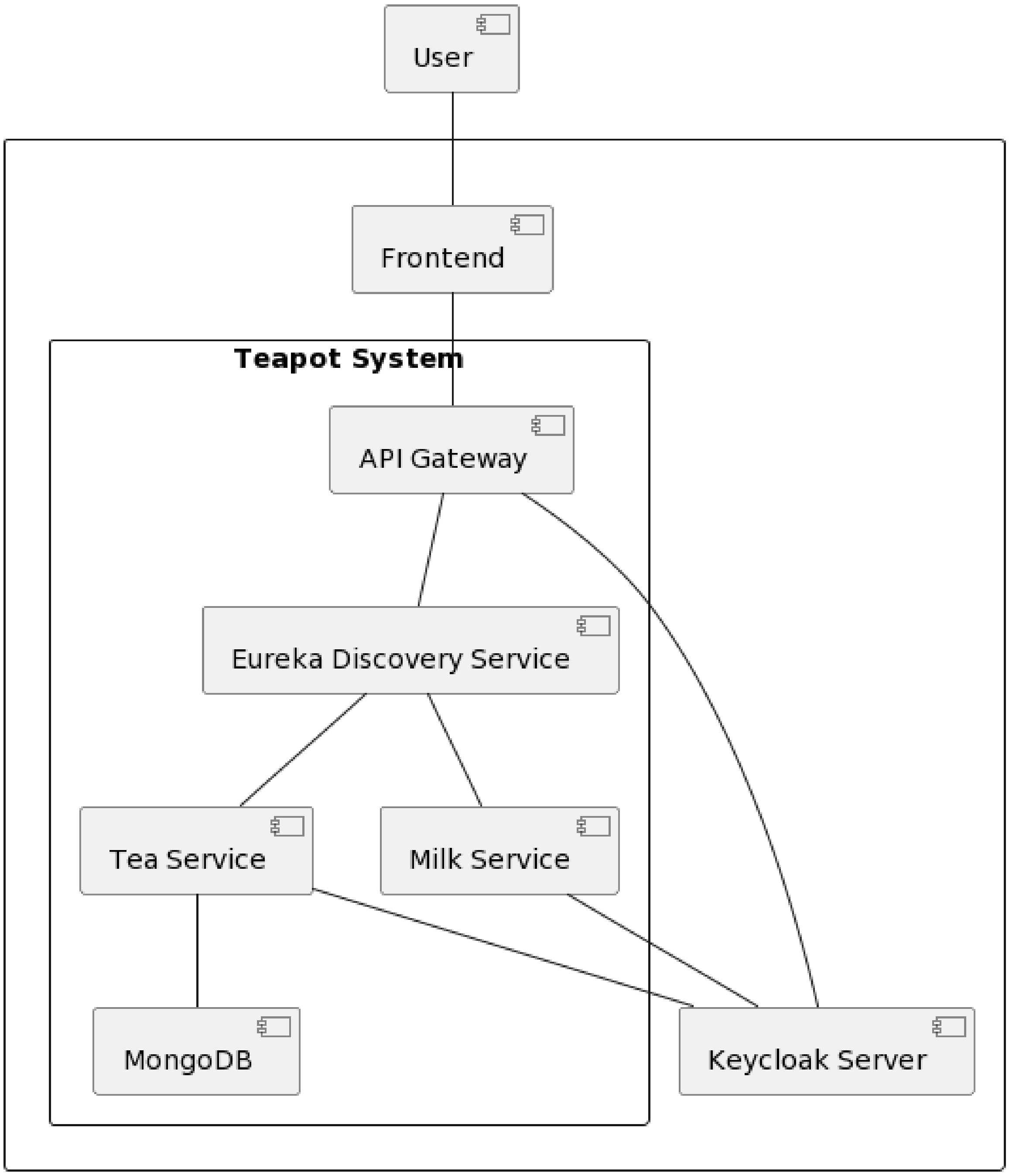


Figure 2.1: High level diagram of the implemented services and their relation to each other

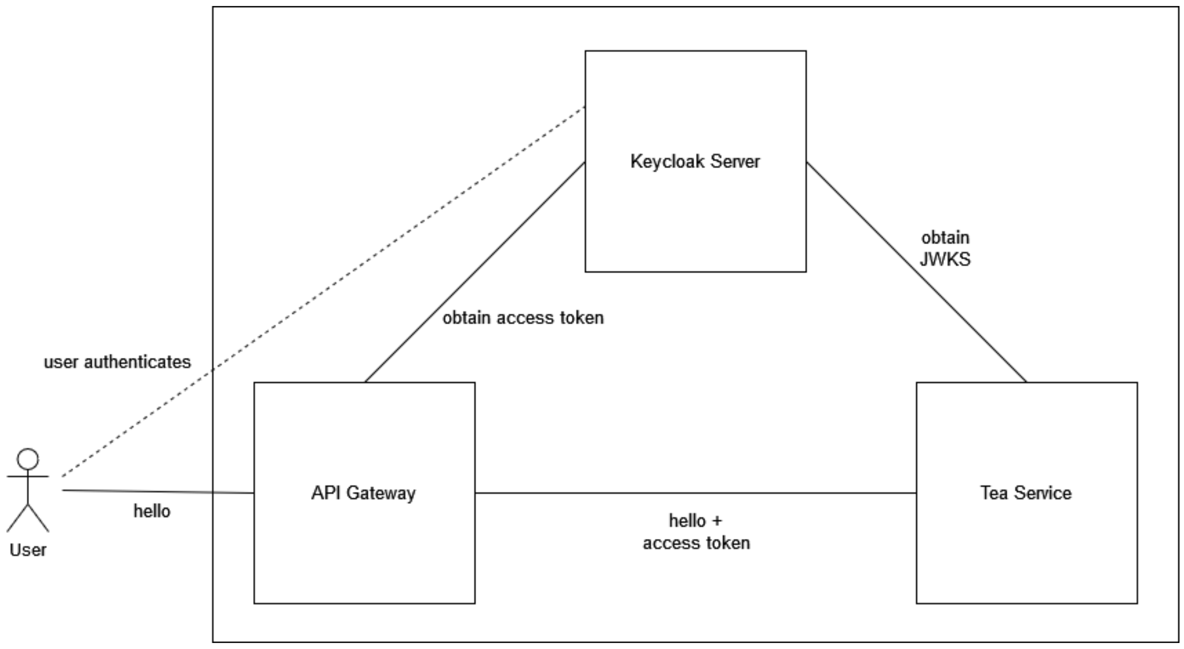


Figure 2.2: High level diagram of the implemented services and their relation to each other

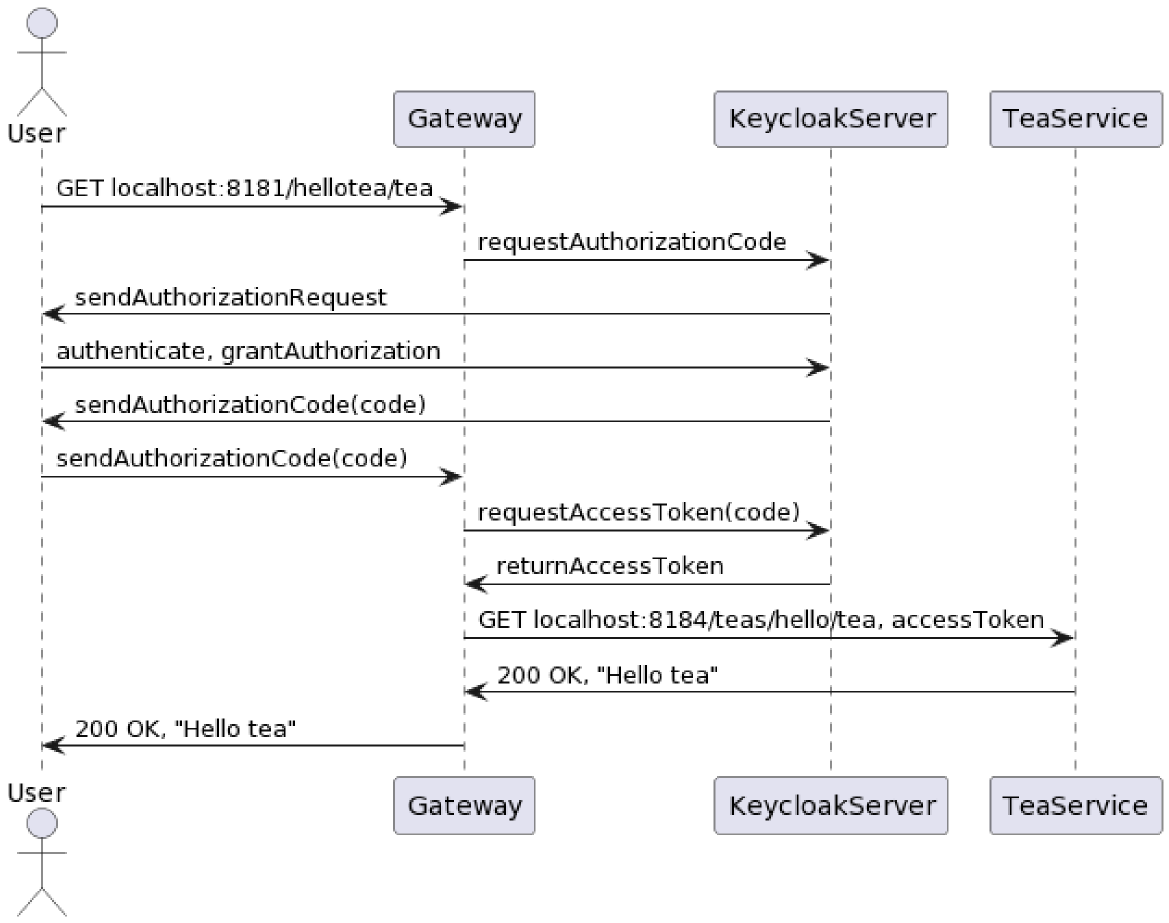

User
Gateway
KeycloakServer
TeaService
GET localhost:8181/hellotea/tea
requestAuthorizationCode
sendAuthorizationRequest
authenticate, grantAuthorization
sendAuthorizationCode(code)
sendAuthorizationCode(code)
requestAccessToken(code)
returnAccessToken
GET localhost:8184/teas/hello/tea, accessToken
200 OK, "Hello tea"
200 OK, "Hello tea"

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

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 recieves 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 Boot[1] 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 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 programmatilally 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 Initializr[2] 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.

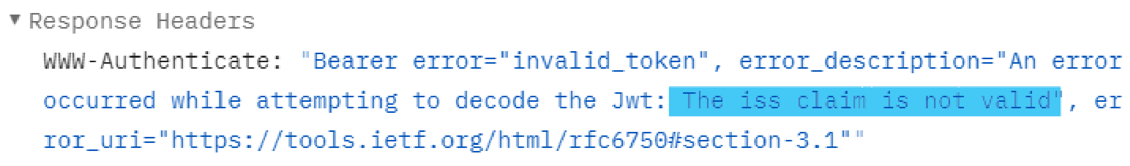
2.2.2 Keycloak Server

Keycloak [**?**] is an identity and access management (IAM) platform. It is open source and published under the Apache Licence 2.0[3]. 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 registrated 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,

[1]https://spring.io/projects/spring-boot
[2]https://start.spring.io/
[3]http://www.apache.org/licenses/LICENSE-2.0

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 it's `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).



```
▼ 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", er
  ror_uri="https://tools.ietf.org/html/rfc6750#section-3.1""
```

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

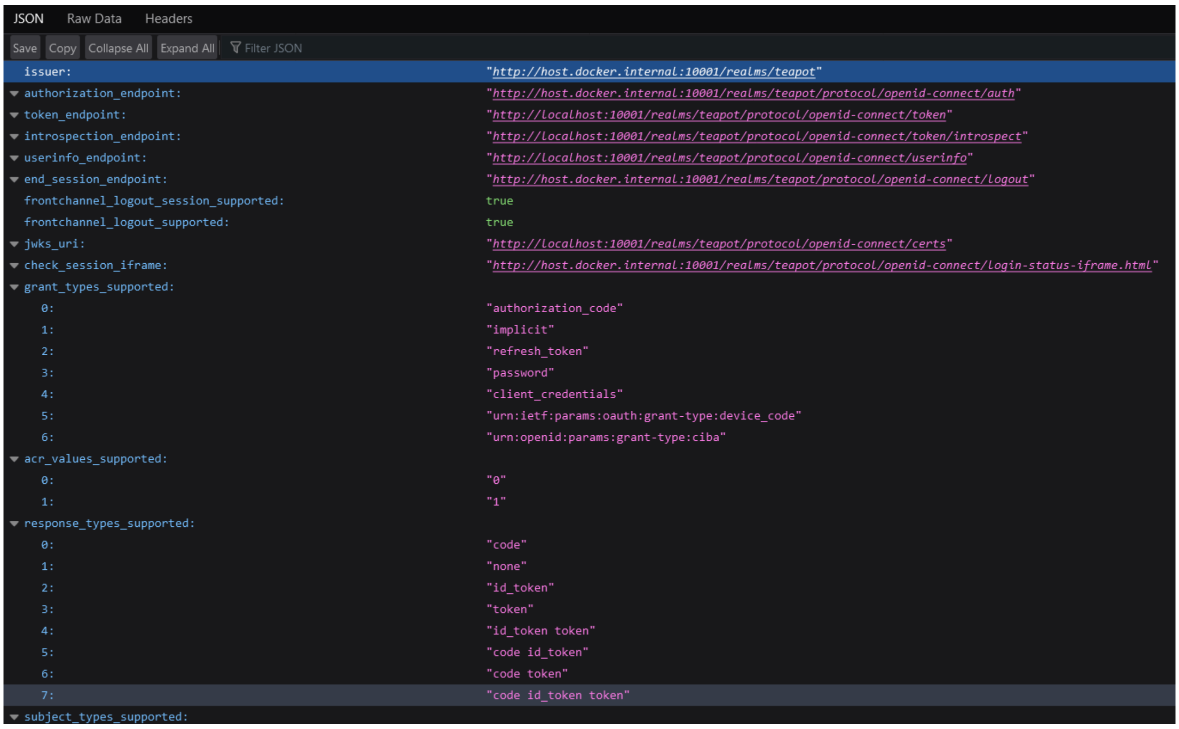
All endpoints of the Keycloak server for a realm are accessible under `<host:port>/realms/<realmnam`
OAuth2 resource servers and clients with the correct `issuer-uri` can call this endpoint to retrieve the other necessary endpoints, like `authorization_endpoint`, `token_endpoint`, `jwks_uri`, etc., but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

2.2.3 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[**?**].



```
issuer:                                 "http://host.docker.internal:10001/realms/teapot"
authorization_endpoint:                 "http://host.docker.internal:10001/realms/teapot/protocol/openid-connect/auth"
token_endpoint:                         "http://localhost:10001/realms/teapot/protocol/openid-connect/token"
introspection_endpoint:                 "http://localhost:10001/realms/teapot/protocol/openid-connect/token/introspect"
userinfo_endpoint:                      "http://localhost:10001/realms/teapot/protocol/openid-connect/userinfo"
end_session_endpoint:                   "http://host.docker.internal:10001/realms/teapot/protocol/openid-connect/logout"
frontchannel_logout_session_supported:  true
frontchannel_logout_supported:          true
jwks_uri:                               "http://localhost:10001/realms/teapot/protocol/openid-connect/certs"
check_session_iframe:                   "http://host.docker.internal:10001/realms/teapot/protocol/openid-connect/login-status-iframe.html"
grant_types_supported:
  0:                                    "authorization_code"
  1:                                    "implicit"
  2:                                    "refresh_token"
  3:                                    "password"
  4:                                    "client_credentials"
  5:                                    "urn:ietf:params:oauth:grant-type:device_code"
  6:                                    "urn:openid:params:grant-type:ciba"
acr_values_supported:
  0:                                    "0"
  1:                                    "1"
response_types_supported:
  0:                                    "code"
  1:                                    "none"
  2:                                    "id_token"
  3:                                    "token"
  4:                                    "id_token token"
  5:                                    "code id_token"
  6:                                    "code token"
  7:                                    "code id_token token"
subject_types_supported:
```

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

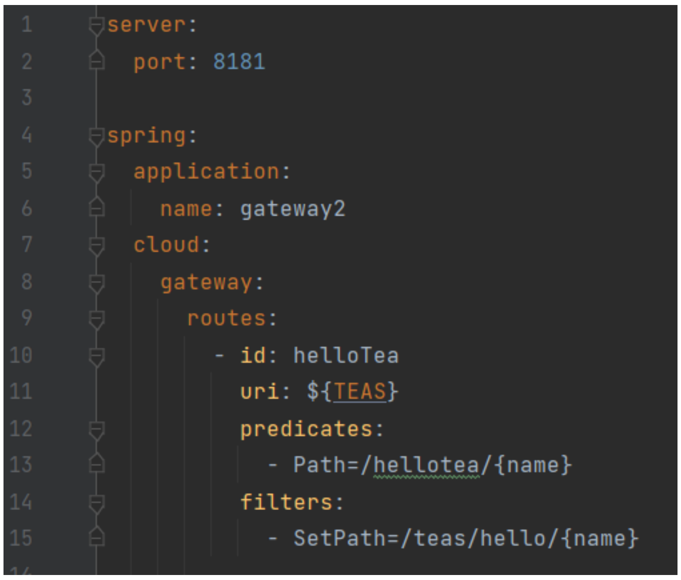
Routes can be configured in the `application.properties` file or in the `application.yml`. Figure 2.6 shows an example route configuration from the `application.yml` file in the reduced Teapot project where no discovery service is used. The `uri` value is given as an environment variable and will be 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/Ula` will be recognized as a match for `$TEAS/teas/hello/Ula`, the path that is set under `filters` with the `SetPath`. `Ula` is an example value for the `name` variable.

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:

```
<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.

```
spring:
```



```yaml
server:
  port: 8181

spring:
  application:
    name: gateway2
  cloud:
    gateway:
      routes:
        - id: helloTea
          uri: ${TEAS}
          predicates:
            - Path=/hellotea/{name}
          filters:
            - SetPath=/teas/hello/{name}
```

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

```yaml
[...]
  security:
    oauth2:
      client:
        provider:
          keycloak-provider:
            issuer-uri: ${keycloak.server-url}/realms/teapot
        registration:
          keycloak-gateway-client:
            provider: keycloak-provider
            scope: openid
            client-id: teapot-gateway
            client-secret: ${client-secret}
            authorization-grant-type: authorization_code
            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 1.4.6, `oidc` 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 1.4). 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` 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` [?].

```yaml
spring:
  application:
    name: gateway2
  cloud:
    gateway:
      routes:

        [...]

        - id: milk
          uri: ${MILK}
          predicates:
            - Path=/milk
          filters:
            - SetPath=/getmilk

      default-filters:
        - 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 [?].

```java
@Configuration
@EnableWebFluxSecurity
public class Gateway2SecurityConfiguration {
    @Bean
    public SecurityWebFilterChain springSecurityWebFilterChain(
          ServerHttpSecurity http,
          ServerLogoutSuccessHandler handler) {
                .authorizeExchange()
                .pathMatchers("/hellogateway", "/hellotea/noauth")
                .permitAll()
```

```
            .and()
                .authorizeExchange()
                .anyExchange()
                .authenticated()
            .and()
                .oauth2Login()
            .and()
                .logout()
                .logoutSuccessHandler(handler);
        return http.build();
    }
```

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

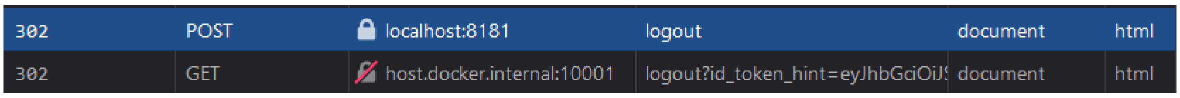
One particular aspect here is the `logoutSuccessHandler` call that gets an `ServerLogoutSuccessHa` object as an argument. A separate bean, as shown in listing **??**, has to be written in order to make this work properly. The `OidcClientInitiatedServerLogoutSuccessHandler`, which implements the ServerLogoutSuccessHandler 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-conn` 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.

```
    @Bean
    public ServerLogoutSuccessHandler keycloakLogoutSuccessHandler(
   ReactiveClientRegistrationRepository repository) {
        OidcClientInitiatedServerLogoutSuccessHandler
   oidcClientInitiatedServerLogoutSuccessHandler = new
   OidcClientInitiatedServerLogoutSuccessHandler(repository);
        oidcClientInitiatedServerLogoutSuccessHandler.
   setPostLogoutRedirectUri("https://orf.at");
        return oidcClientInitiatedServerLogoutSuccessHandler;
    }
```

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

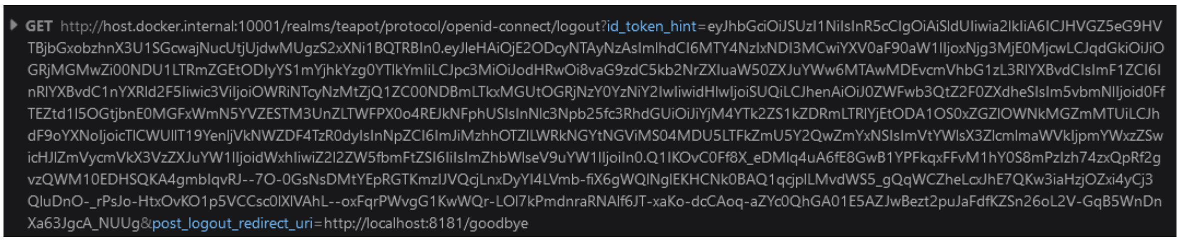
With Spring Boot, a `GatewayApplication.java` class is created automatically, that contains the `main` method. With this setup the Gateway application is already fully functional and able to route requests to a resource server together with an access token after the user has authenticated successfully.

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-



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

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



```
▸ GET http://host.docker.internal:10001/realms/teapot/protocol/openid-connect/logout?id_token_hint=eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJHVGZ5eG9HV
TBjbGxobzhnX3U1SGcwajNucUtjUjdwMUgzS2xXNi1BQTRBIn0.eyJleHAiOjE2ODcyNTAyNzAsImlhdCI6MTY4NzIxNDI3MCwiYXV0aF90aW1lIjoxNjg3MjE0MjcwLCJqdGkiOiJiO
GRjMGMwZi00NDU1LTRmZGEtODIyYS1mYjhkYzg0YTlkYmIiLCJpc3MiOiJodHRwOi8vaG9zdC5kb2NrZXIuaW50ZXJuYWw6MTAwMDEvcmVhbG1zL3RlYXBvdCIsImF1ZCI6I
nRlYXBvdC1nYXRld2F5Iiwic3ViIjoiOWRiNTcyNzMtZjQ1ZC00NDBmLTkxMGUtOGRjNzY0YzNiY2IwIiwidHlwIjoiSUQiLCJhenAiOiJ0ZWFwb3QtZ2F0ZXdheSIsIm5vbmNlIjoid0Ff
TEZtd1I5OGtjbnE0MGFxWmN5YVZESTM3UnZLTWFPX0o4REJkNFphUSIsInNlc3Npb25fc3RhdGUiOiJiYjM4YTk2ZS1kZDRmLTRlYjEtODA1OS0xZGZlOWNkMGZmMTUiLCJh
dF9oYXNoIjoicTlCWUlIT19YenljVkNWZDF4TzR0dyIsInNpZCI6ImJiMzhhOTZlLWRkNGYtNGViMS04MDU5LTFkZmU5Y2QwZmYxNSIsImVtYWlsX3ZlcmlmaWVkIjpmYWxzZSw
icHJlZmVycmVkX3VzZXJuYW1lIjoidWxhIiwiZ2l2ZW5fbmFtZSI6IiIsImZhbWlseV9uYW1lIjoiIn0.Q1IKOvC0Ff8X_eDMlq4uA6fE8GwB1YPFkqxFFvM1hY0S8mPzIzh74zxQpRf2g
vzQWM10EDHSQKA4gmbIqvRJ--7O-0GsNsDMtYEpRGTKmzIJVQcjLnxDyYI4LVmb-fiX6gWQINgIEKHCNk0BAQ1qcjplLMvdWS5_gQqWCZheLcxJhE7QKw3iaHzjOZxi4yCj3
QIuDnO-_rPsJo-HtxOvKO1p5VCCsc0lXlVAhL--oxFqrPWvgG1KwWQr-LOI7kPmdnraRNAlf6JT-xaKo-dcCAoq-aZYc0QhGA01E5AZJwBezt2puJaFdfKZSn26oL2V-GqB5WnDn
Xa63JgcA_NUUg&post_logout_redirect_uri=http://localhost:8181/goodbye
```

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.

```java
@RestController
public class GatewayController {
    @GetMapping("/hellogateway")
    public String greet(@AuthenticationPrincipal OAuth2User principal) {
        return "Hello, " + principal.getName() + ", from Gateway";
    }
}
```

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

2.2.4 The Resource Server

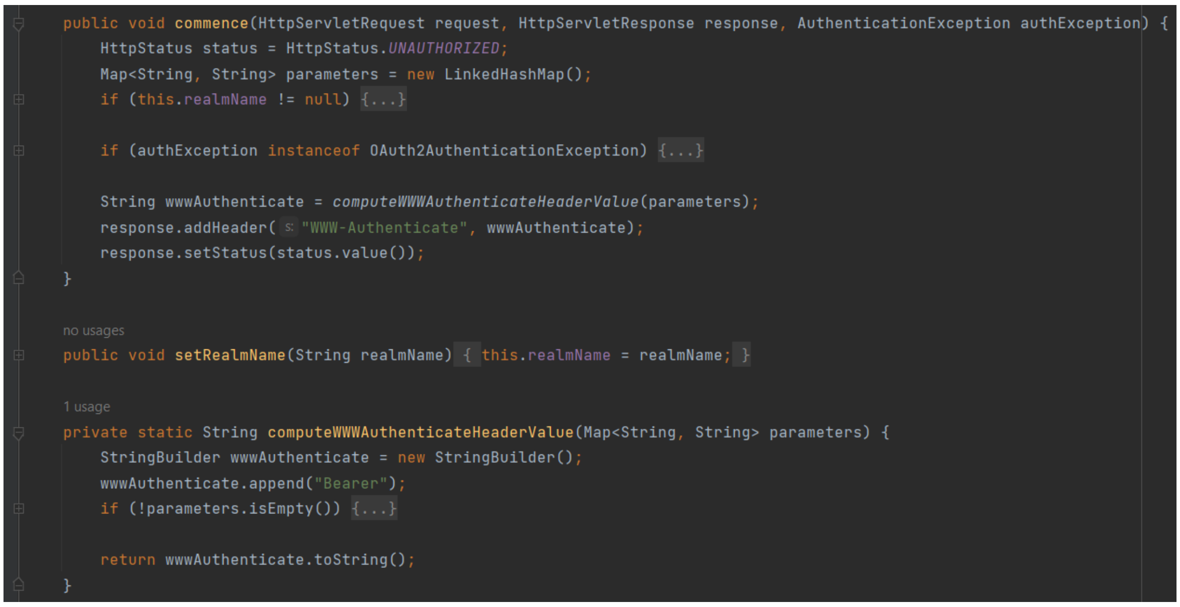
The OAuth2 Resource Server recieves and validates the access token and, if the token is valid, grants access to the requested resource (see section 1.4). The basic steps to configure a resource server with Spring Boot are not very different from the configuration of the OAuth2 client: implementation of the necessary dependencies in the `pom.xml` file, configuration of the `issure-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 [?]:

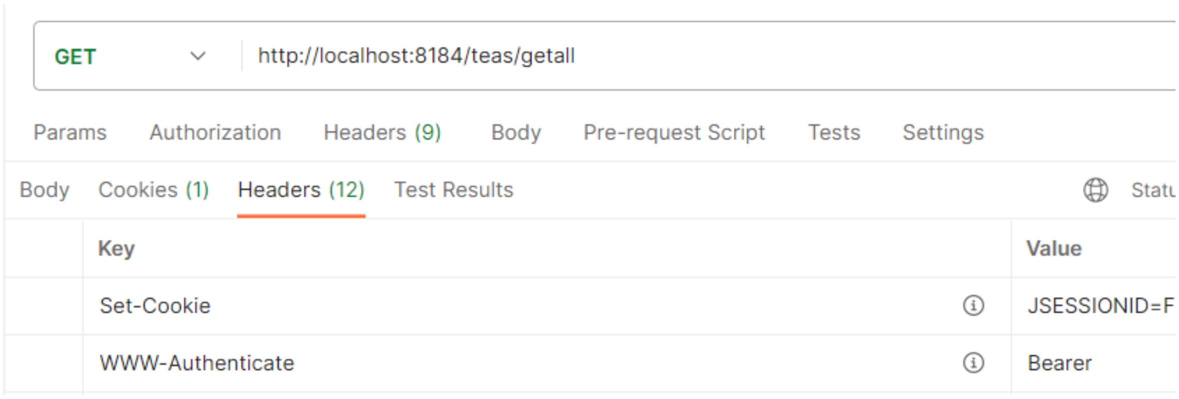
- An unauthenticated request comes in from the User
- The `AuthorizationFilter` throws an `AccessDeniedException`

- 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.



```
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( s: "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

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

When the request comes with a bearer token, the `BearerTokenAuthenticationFilter` extracts the token from the `HttpServletRequest` and creates a `BearerTokenAuthenticationToken,` 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 [?], crentials 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 jkw-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).

```
spring:

[...]

  security:
    oauth2:
      resource-server:
        jwt:
          issuer-uri: ${KEYCLOAK}/realms/teapot
          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 overridden 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 [?].

```java
@Configuration
public class TeaSecurityConfiguration {
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
   Exception {
        http
                .authorizeHttpRequests().requestMatchers("/teas/hello/
   noauth", "/teas/create").permitAll()
                .anyRequest().authenticated();
        http.oauth2ResourceServer(OAuth2ResourceServerConfigurer::jwt);
        [...]
        return http.build();
    }
}
```

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.

```java
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws
   Exception {
        [...]
        http.sessionManagement((session) -> session.sessionCreationPolicy
   (SessionCreationPolicy.STATELESS))
                .csrf().disable();
        return http.build();
    }
```

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

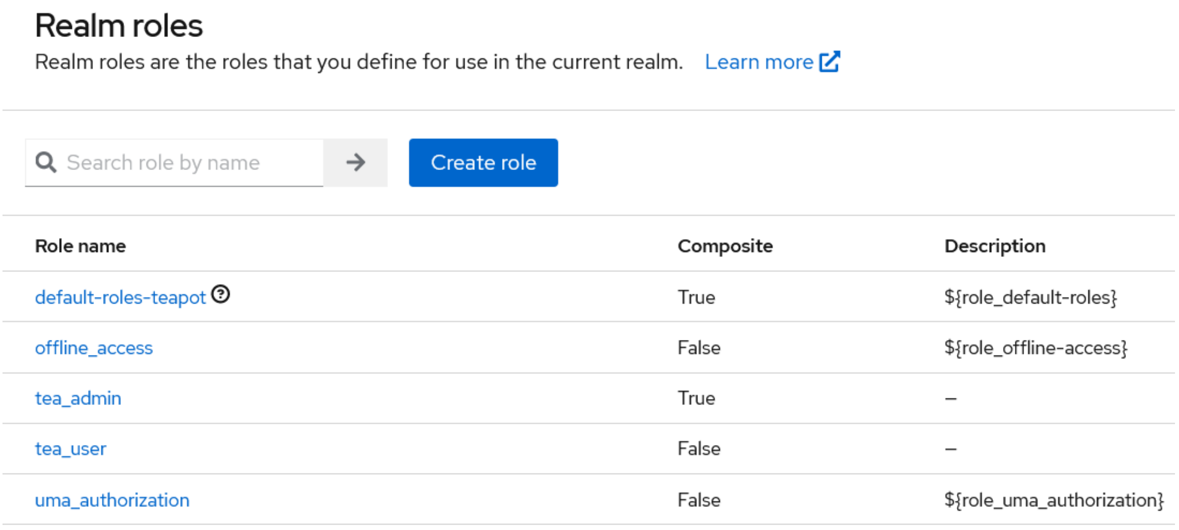
The actual resources that the Tea service is holding are `Tea` objects that are stored in a MongoDB database. However, the implementation of object relational mapping with spring boot lies beyond the scope of this thesis, therefore further details are omitted.

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 possibile 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

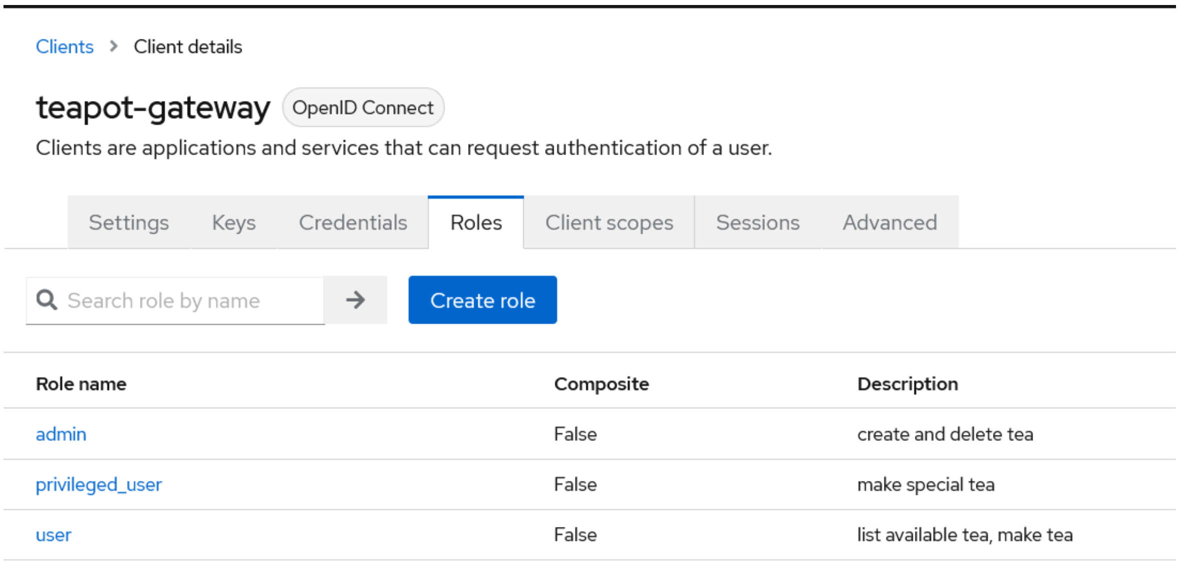
Role name	Composite	Description
default-roles-teapot	True	${role_default-roles}
offline_access	False	${role_offline-access}
tea_admin	True	–
tea_user	False	–
uma_authorization	False	${role_uma_authorization}

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 it's associated client roles, `admin`,
texttuser 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` 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`.

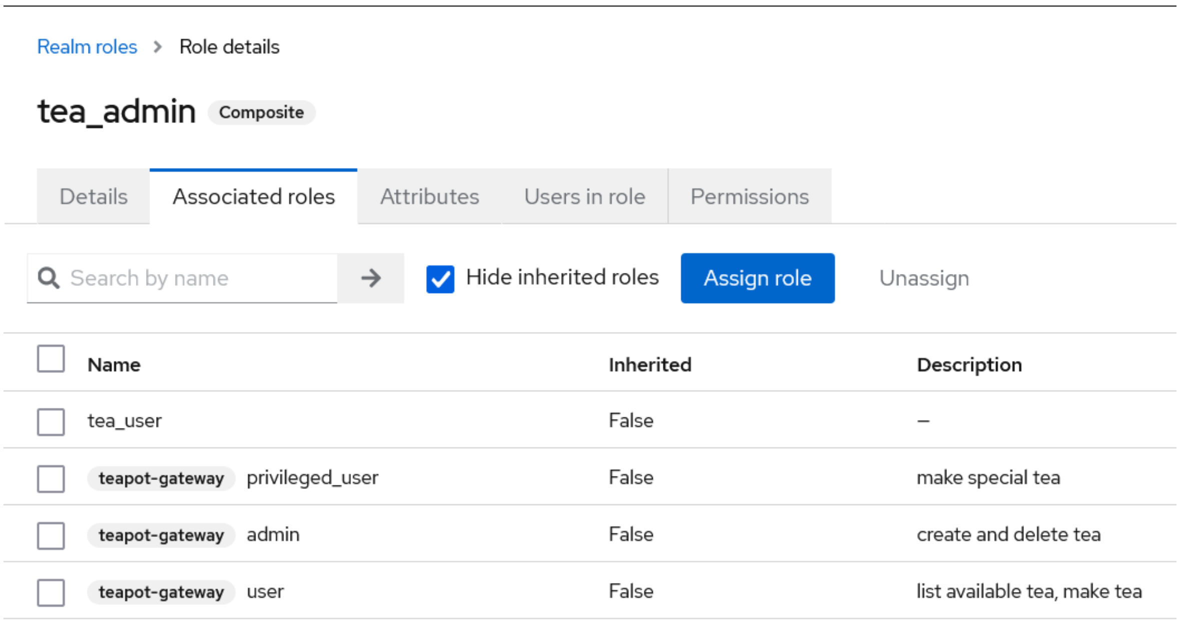
As we can see, the access token does not contain `authority` claims. The `realm_access` or `resource_access` access claims are specific to Keycloak (!!! Quelle). Spring Security allows to check for authorities inside the `authentication` object, but it provides no means by default to check specifically for the access claims provided by a Keycloak access



Clients are applications and services that can request authentication of a user.

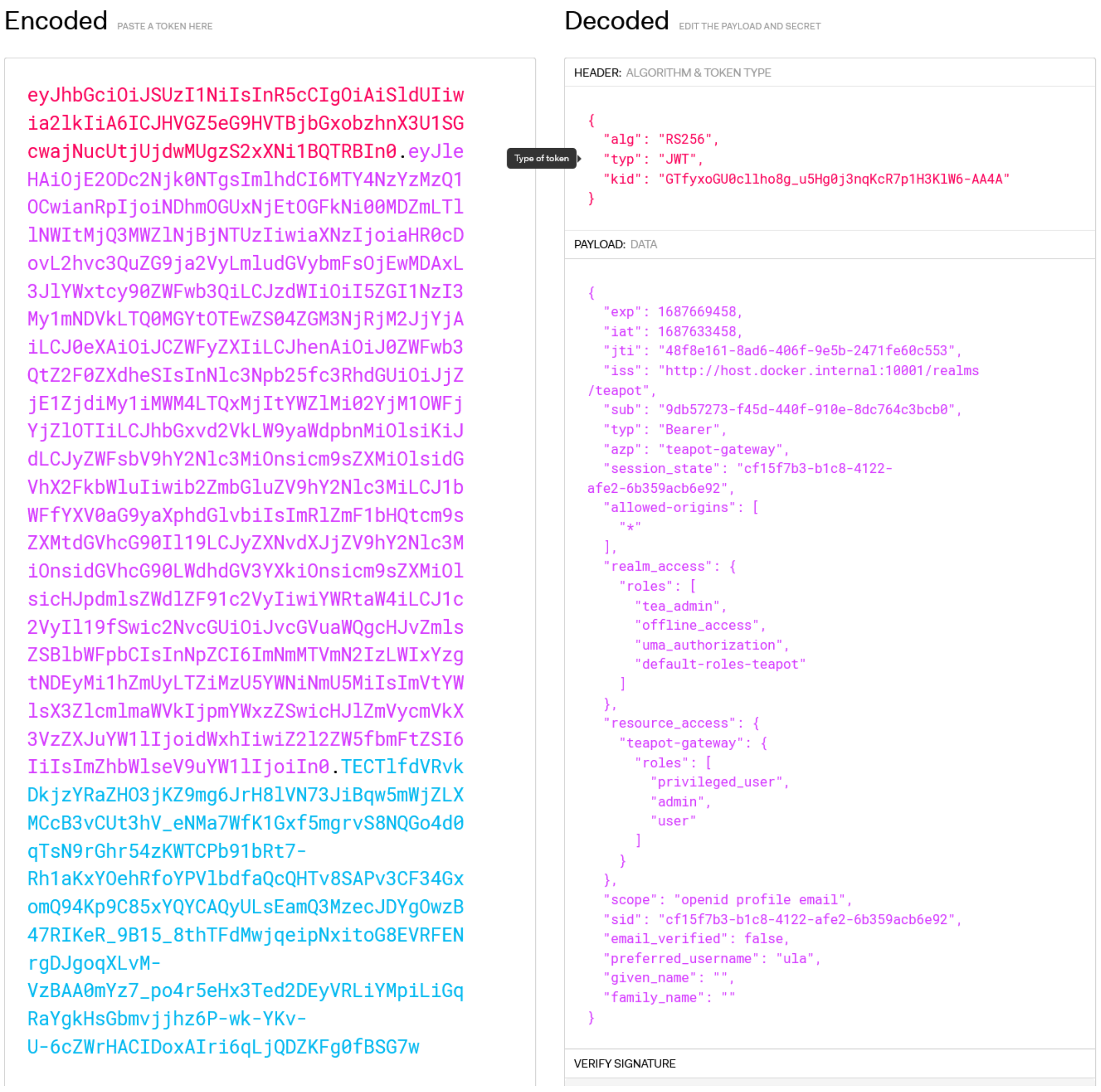
Role name	Composite	Description
admin	False	create and delete tea
privileged_user	False	make special tea
user	False	list available tea, make tea

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



	Name	Inherited	Description
☐	tea_user	False	–
☐	teapot-gateway privileged_user	False	make special tea
☐	teapot-gateway admin	False	create and delete tea
☐	teapot-gateway user	False	list available tea, make tea

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


Encoded PASTE A TOKEN HERE
Type of token
Decoded EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE
PAYLOAD: DATA
VERIFY SIGNATURE

```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiw
ia2lkIiA6ICJHVGZ5eG9HVTBjbGxobzhnX3U1SG
cwajNucUtjUjdwMUgzS2xXNi1BQTRBIn0.eyJle
HAiOjE2ODc2Njk0NTgsImlhdCI6MTY4NzYzMzQ1
OCwianRpIjoiNDhmOGUxNjEtOGFkNi00MDZmLTl
lNWItMjQ3MWZlNjBjNTUzIiwiaXNzIjoiaHR0cD
ovL2hvc3QuZG9ja2VyLmludGVybmFsOjEwMDAxL
3JlYWxtcy90ZWFwb3QiLCJzdWIiOiI5ZGI1NzI3
My1mNDVkLTQ0MGYtOTEwZS04ZGM3NjRjM2JjYjA
iLCJ0eXAiOiJCZWFyZXIiLCJhenAiOiJ0ZWFwb3
QtZ2F0ZXdheSIsInNlc3Npb25fc3RhdGUiOiJjZ
jE1ZjdiMy1iMWM4LTQxMjItYWZlMi02YjM1OWFj
YjZlOTIiLCJhbGxvd2VkLW9yaWdpbnMiOlsiKiJ
dLCJyZWFsbV9hY2Nlc3MiOnsicm9sZXMiOlsidG
VhX2FkbWluIiwib2ZmbGluZV9hY2Nlc3MiLCJ1b
WFfYXV0aG9yaXphdGlvbiIsImRlZmF1bHQtcm9s
ZXMtdGVhcG90Il19LCJyZXNvdXJjZV9hY2Nlc3M
iOnsidGVhcG90LWdhdGV3YXkiOnsicm9sZXMiOl
sicHJpdmlsZWdlZF91c2VyIiwiYWRtaW4iLCJ1c
2VyIl19fSwic2NvcGUiOiJvcGVuaWQgcHJvZmls
ZSBlbWFpbCIsInNpZCI6ImNmMTVmN2IzLWIxYzg
tNDEyMi1hZmUyLTZiMzU5YWNiNmU5MiIsImVtYW
lsX3ZlcmlmaWVkIjpmYWxzZSwicHJlZmVycmVkX
3VzZXJuYW1lIjoidWxhIiwiZ2l2ZW5fbmFtZSI6
IiIsImZhbWlseV9uYW1lIjoiIn0.TECTlfdVRvk
DkjzYRaZHO3jKZ9mg6JrH8lVN73JiBqw5mWjZLX
MCcB3vCUt3hV_eNMa7WfK1Gxf5mgrvS8NQGo4d0
qTsN9rGhr54zKWTCPb91bRt7-
Rh1aKxYOehRfoYPVlbdfaQcQHTv8SAPv3CF34Gx
omQ94Kp9C85xYQYCAQyULsEamQ3MzecJDYgOwzB
47RIKeR_9B15_8thTFdMwjqeipNxitoG8EVRFEN
rgDJgoqXLvM-
VzBAA0mYz7_po4r5eHx3Ted2DEyVRLiYMpiLiGq
RaYgkHsGbmvjjhz6P-wk-YKv-
U-6cZWrHACIDoxAIri6qLjQDZKFg0fBSG7w
```

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "GTfyxoGU0cllho8g_u5Hg0j3nqKcR7p1H3KlW6-AA4A"
}
```

```
{
  "exp": 1687669458,
  "iat": 1687633458,
  "jti": "48f8e161-8ad6-406f-9e5b-2471fe60c553",
  "iss": "http://host.docker.internal:10001/realms
/teapot",
  "sub": "9db57273-f45d-440f-910e-8dc764c3bcb0",
  "typ": "Bearer",
  "azp": "teapot-gateway",
  "session_state": "cf15f7b3-b1c8-4122-
afe2-6b359acb6e92",
  "allowed-origins": [
    "*"
  ],
  "realm_access": {
    "roles": [
      "tea_admin",
      "offline_access",
      "uma_authorization",
      "default-roles-teapot"
    ]
  },
  "resource_access": {
    "teapot-gateway": {
      "roles": [
        "privileged_user",
        "admin",
        "user"
      ]
    }
  },
  "scope": "openid profile email",
  "sid": "cf15f7b3-b1c8-4122-afe2-6b359acb6e92",
  "email_verified": false,
  "preferred_username": "ula",
  "given_name": "",
  "family_name": ""
}
```

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

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 by [**?**] 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.

```java
@RequiredArgsConstructor
class JwtGrantedAuthoritiesConverter implements Converter<Jwt, Collection<? extends GrantedAuthority>> {

    @Override
    @SuppressWarnings({"rawtypes", "unchecked"})
    public Collection<? extends GrantedAuthority> convert(Jwt jwt) {
        return Stream.of("$.realm_access.roles", "$.resource_access.*.roles").flatMap(claimPaths -> {
                        Object claim;
                        try {
                            claim = JsonPath.read(jwt.getClaims(), claimPaths);
                        } catch (PathNotFoundException e) {
                            return Stream.empty();
                        }
                        final var firstItem = ((Collection) claim).iterator().next();
                        if (firstItem instanceof String) {
                            return (Stream<String>) ((Collection) claim).stream();
                        }
                        if (Collection.class.isAssignableFrom(firstItem.getClass())) {
                            return (Stream<String>) ((Collection) claim).stream().flatMap(item -> ((Collection) item).stream()).map(String.class::cast);
                        }
                        return Stream.empty();
                    })
                    .map(authority -> new SimpleGrantedAuthority("ROLE_" + authority))
                    .map(GrantedAuthority.class::cast).toList();
    }
}
}
```

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.

```java
@Component
@RequiredArgsConstructor
class SpringAddonsJwtAuthenticationConverter implements Converter<Jwt, JwtAuthenticationToken> {

    @Override
    public JwtAuthenticationToken convert(Jwt jwt) {
        final var authorities = new JwtGrantedAuthoritiesConverter().convert(jwt);
        final String username = JsonPath.read(jwt.getClaims(), "preferred_username");
        return new JwtAuthenticationToken(jwt, authorities, username);
    }
}
```

Listing 2.10: Custom converter to set the extracted granted authorities from the access token in the new `Authentication` [?]

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 corresondent authority must be allowed explicitly in addition to the respective user roles (see lines 21-25 in figure 2.2.5).

```java
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     @Bean
12     public SecurityFilterChain filterChain(HttpSecurity http, Converter<Jwt, ? extends AbstractAuthenticationToken> jwtAuthenticationConverter) throws Exception {
13         http
14                 .authorizeHttpRequests().requestMatchers("/teas/hello/noauth")
15                 .permitAll()
16                 .requestMatchers("/teas/admin")
```

```
                .hasRole(REALM_ADMIN)
                .requestMatchers("/teas/create", "/teas/delete/*")
                .hasRole(CLIENT_ADMIN)
                .requestMatchers("/teas/maketea/special")
                .hasAnyRole(CLIENT_PRIVILEGED_USER, CLIENT_ADMIN)
                .requestMatchers("/teas/maketea/*", "/teas/hello/user")
                .hasAnyRole(CLIENT_USER, CLIENT_PRIVILEGED_USER,
    CLIENT_ADMIN)
                .requestMatchers("/teas/getall")
                .hasAnyRole(REALM_USER, REALM_ADMIN, CLIENT_ADMIN)
            .anyRequest().authenticated();
        http.oauth2ResourceServer().jwt().jwtAuthenticationConverter(
    jwtAuthenticationConverter);
        http.sessionManagement().sessionCreationPolicy(
    SessionCreationPolicy.STATELESS);
        http.csrf().disable();
        return http.build();
    }
}
```

2.3 Load testing with JMeter

2.4 Code Analysis

LoC -> Dependencies? + Sonarqube Ergebnisse?

3 Discussion (Results in Implementation integrieren?)

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

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, - testing

List of Figures

Listings

List of Tables