FH CAMPUS WIEN
UNIVERSITY OF APPLIED SCIENCES

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

ALTER ABSTRACT! STEHT NOCH HIER ALS HINT FÜR DEN LINEBREAK 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
OIDC	OpenID Connect
RFC	Request for Comments
POC	Proof of Concept
RSA	Rivest-Shamir-Adelman
SHA	Secure Hash Algorithm
SSO	Single Sign-on
XACML	Extensible Access Control Markup Language

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

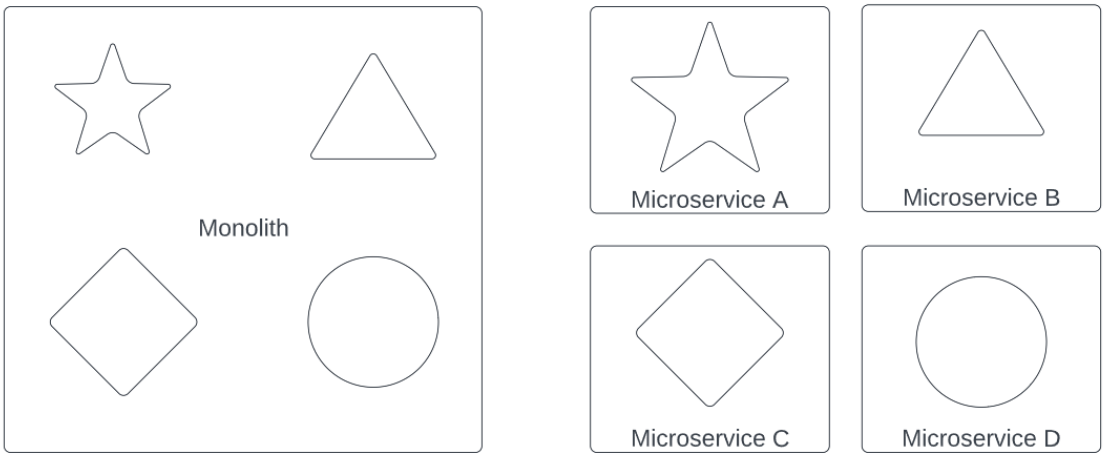

Figure 1.1: A very abstract illustration of a monolith and micoservices according to [**?**]

1.3 MSA Security Challenges

!!! weglassen?

1.4 Authentication and Authorization

!!! ALT - Neu formulieren As we have seen in the previous sections, when talking about security in microservices, both, authentication and authorization are considered to be among

the most important topics. In this section, a clarification is given how the two concepts differ and what their role is in the context of MSA.

Authentication and authorization, although they sound very similar, are two distinctive concepts and it is important to understand the difference between the two. To put it very shortly, authentication is about identity and authorization is about permissions. When someone authenticates, they usually provide a proof of the fact that what they say who they are is true. Identity can be proven with something they know (e.g., a secret password), something they have (like a phone number, to which a code can be sent), or something they are (like biometric data), or a combination of those (e.g. in two-factor authentication where a user gives their username and password, but the authentication is only complete when they also give a code that was sent to their phone number or e-mail address). Not only a person can prove their identity. A system can authenticate as well, for example with a secret or certificate or both. Authorization on the other hand deals with the question what a person or entity is allowed to do, for example where they can enter. Because this usually depends on who they are, in order to determine which permissions someone or something has, authentication is necessary first. Siriwardena [**?**, p. 133] gives a visa control at a border as example: A person who wants to cross the border has to authenticate - their picture and/or fingerprint might be validated as truly belonging to that person, perhaps also by comparing to a database. This is authentication. Knowing who that person is does not get this person across the border, unless they have a visa. The visa has to be valid and not expired and can contain further details about what you are allowed to do in that country [**?**, p. 133].

Now, what is special about authentication and authorization in a MSA? In a monolithic architecture, when a user wants to access a resource, they authenticate with the system and might get back a session token, which can be sent with every subsequent request to the system. The token can then be validated inside the system, and it will know if this user has the necessary permission for the requested transaction. The same process becomes way more complex in MSA. While a monolithic application only has few entry points, a MSA has at least as many entry points as deployed services and each of them should be protected [**?**], [**?**]. When a requested resource or service does not sit inside the same system as the service where the user has authenticated themselves, the first service does not know if it can trust whoever made this request without either making another request to the server responsible for dealing with authentication and permissions, or having a way to validate the token locally, without further communication, which might not always be possible. This results in a higher number of requests between services and therefore in potentially decreased performance of the MSA.

To make everything worse, not only access by external end-users has to be managed, but also communication between services. Although the MSA principle demands loosely coupled microservices, it is sometimes inevitable that one service has to talk to another service in order to fulfil its job. But a service should not be reachable by any other microservices in the system, only by those who have a good reason to do so [**?**]. When a service talks to another service on behalf of a user, the user information (authentication and authorization) can be passed on down the line, so each microservice knows who they are working for. This is called principal propagation [**?**].

1.5 Authentication Patterns

!!! Backend for Frontend vs. Frontend als Client + Diagramme. Wireshark eher erst unter implementation?

1.6 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.

1.6.1 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 [**?**].

1.6.2 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

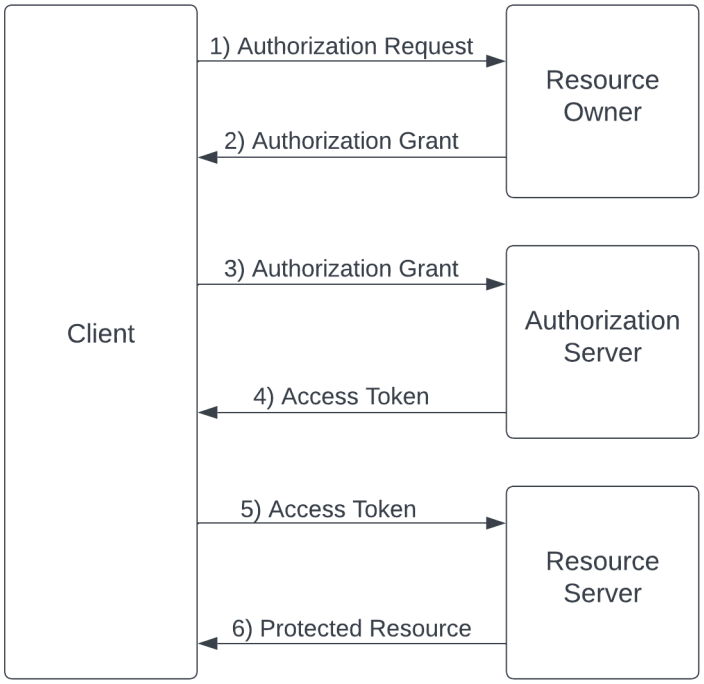

Figure 1.2: Abstraction of the OAuth2 flow after [**?**, fig. 1]

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.

1.6.3 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.

1.6.4 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.

1.6.5 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 [?]

1.6.6 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 1.6.3, 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 [?].

1.7 Methodology

Für Forschungsfrage 1: Follow Spring Boot Documentation and Tutorials -> implement the Teapot and test Authentication and Authorization functionality with Browser, Postman and Wireshark.

Für Forschungsfrage 2: Implement a reduced version of the Teapot system in three different ways and perform load tests with jmeter. Compare response times.

2 Implementation

!!! hier nur abgelegt: With the basic implementation of the prototype system (see section 4 Impl (reihenfolge tauschen?), 4 different versions were created: one with the Gateway as the OAuth2 client and with the Tea Service as a Resource Server, one where the Gateway and the Tea Service are both Resource Servers and one where only the Gateway is a Resource Server and the Tea Service remains unprotected. The initial intention was to implement MTLS between the Gateway and the Resource Server as recommended in [Siriwardena - Microservices Security in Action - Seite???]. This last version was later abandoned in favour of the focus on the difference of the client position in the system. Testing a difference between OAuth2 and MTLS lies outside the scope of this thesis.

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 in order 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.

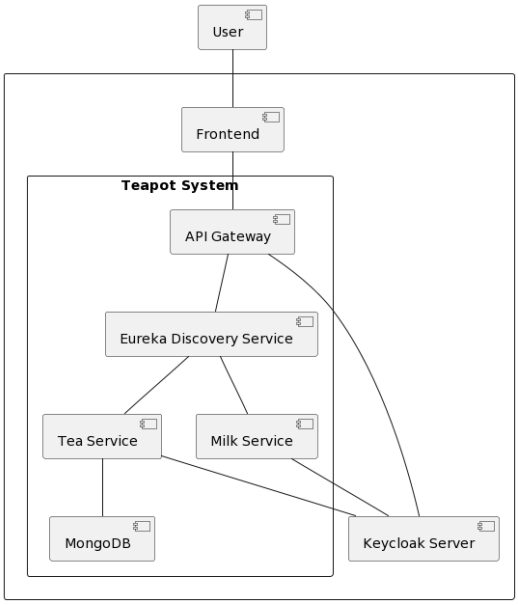

Figure 2.1: High level diagram of the implemented services and their relation to each other

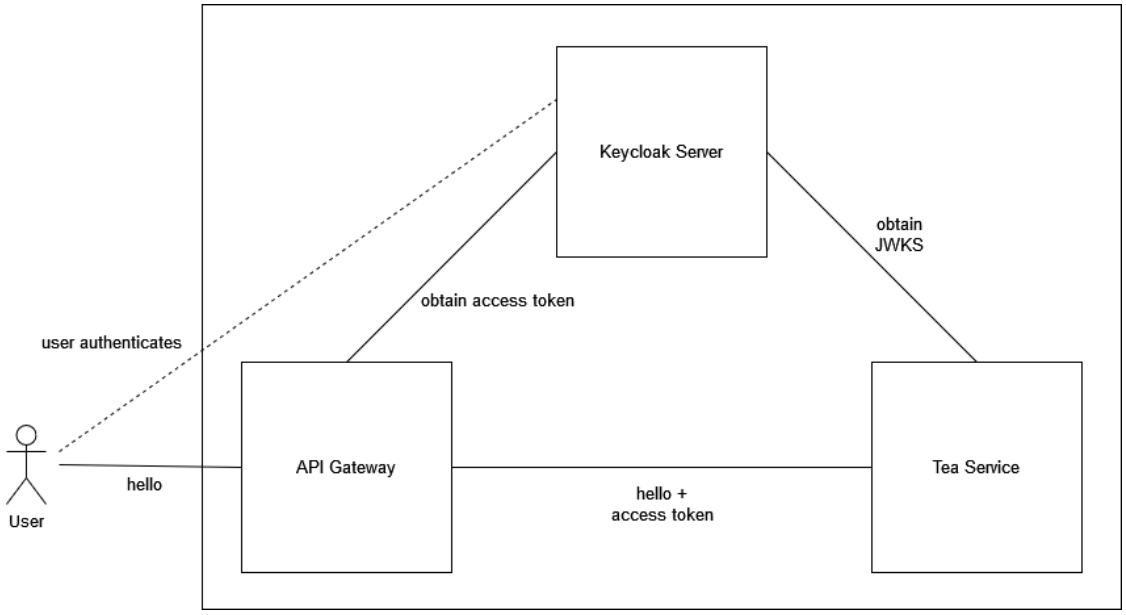

Figure 2.2: High level diagram of the implemented services and their relation to each other

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

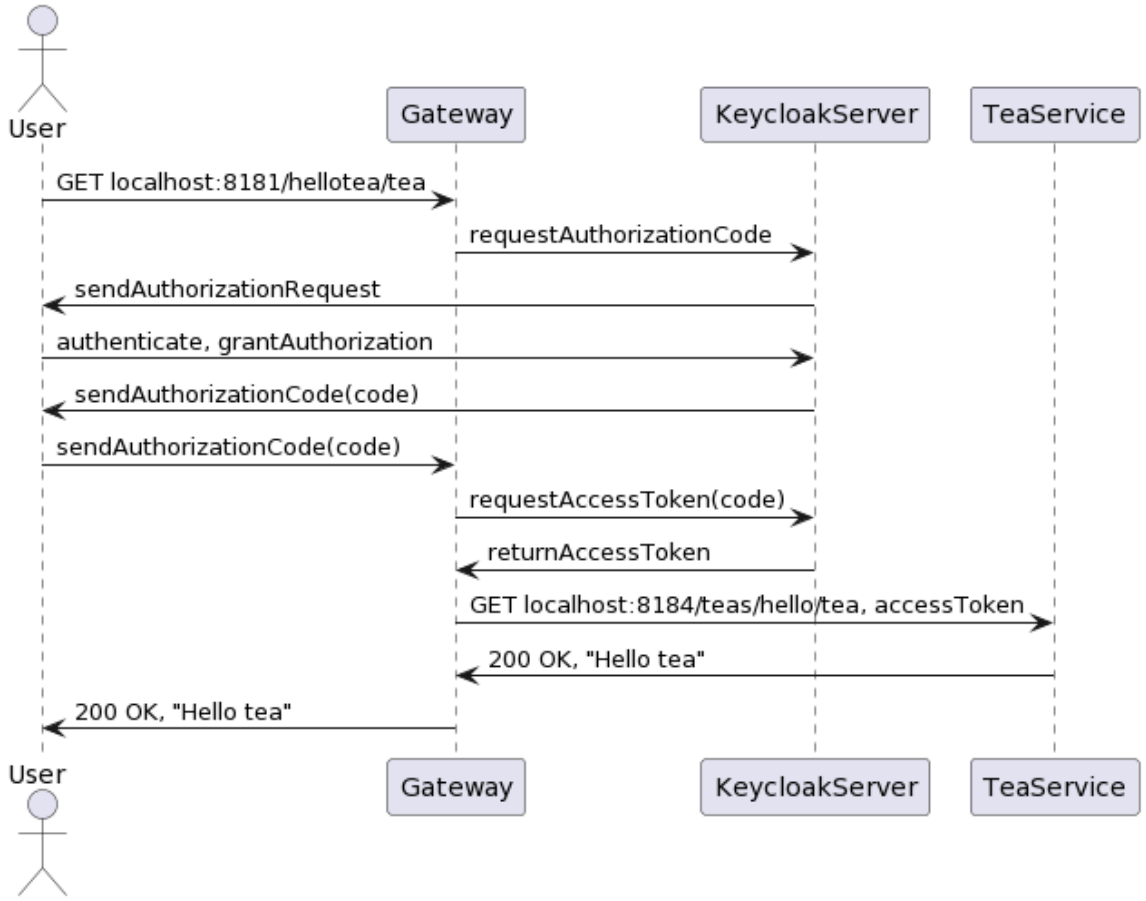

Figure 2.3: Sequence diagram of the first request to a protected resource including a simplified auth code grant flow

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

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 for spring and starter dependencies that bundle selections of libraries in one Maven or Gradle dependency [**?**, pp. 4]. These help to reduce the need for the developer to write boilerplate code or configurations manually, which means that one big advantage when using Spring boot is the quick project setup and the fact that it brings many features production-ready out of the box, like logging, monitoring, security, and error handling. The widely used `spring-boot-starter-web` module also has TomCat webserver automatically embedded (for reactive web applications there is the `spring-boot-starter-webflux` module which comes with Netty as default). However, all configurations can be overridden or customized when needed, either programmatilally with Java or by adding configurations to the `applications.properties` or the `applications.yml` file. For the Teapot project the `applications.yml` file 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 created with the Spring Boot Initializr [!!!link...] which is also available when creating a new Project in IntelliJ. All Maven dependencies or starter modules 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 Spring Cloud Gateway

The Gateway's job in a MSA is to route requests to services beyond. There is a special Spring Boot module, the `spring-cloud-starter-gateway` module, which 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 route-specific 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 2.4 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

[1] https://spring.io/projects/spring-boot

as an environment variable and injected via the docker compose.yml file (!!! see docker). With the discovery service, the value would be `lb://` followed by the name that the Tea service application uses to register with the Eureka discovery service. In 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`. Note that `Ula` is an example value for the `name` variable.

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

In order to configure the Gateway as a OAuth2 client, we also need to include the `spring-boot-starter-oauth2-client` starter module 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 module and to not get confused by the different oauth2-client dependencies available. The `spring-boot-starter-oauth2-client` module is intended to be used with Spring Boot. Then, after having created the client in Keycloak (see section 2.2.3), 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.properties` or `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 1.6.6, `oidc` must be included as 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 1.6).

Security configuration can now be added in the way that is shown in the following code example, taken from the reduced Teapot Gateway2:

```
@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();
    }
```

!!! Erklären!

The Gateway must also be able to attach access tokens to any authorized request that will be routed to a 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 enable the token relay is certainly to add a `default-filter` to the route configuration in the `application.yml` file as shown in listing 2.1.

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

        [...]

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

      default-filters:
        - TokenRelay=
```

Listing 2.1: Route configuration with token relay default filter in the Gateway's application.yml file

With Spring Boot, an `GatewayApplication.java` class that contains the `main` method is created automatially. 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 Gateway is the `/hellogateway` endpoint which returns a string with a greeting to the user after reading the user's name from the authentication principal. For the ability to

unterschiede zwischen den verschiedenen client-versionen mit trial + error rausgefunden oder gibts einen link dazu?

2.2.3 Keycloak Server

Was ist das docker admin console realms clients users wie hab ich ihn configuriert? Docker compose -> config importiert

3 Results and Discussion

3.1 Response times

JMeter Results

3.2 Code Analysis

LoC -> Dependencies? + Sonarqube Ergebnisse?

4 Conclusion and Future Work

List of Figures

List of Tables