Implementation of OAuth 2 and OpenID Connect in a microservice-based prototype with Spring Boot 3

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:
Mag^{a} Ursula Rauch

Student identification number:
00514397

Supervisor:
Leon Freudenthaler, BSc MSc

Date:
10.08.2023

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: 10.08.2023 Signature:

Acknowledgement

Viele Personen haben mich im Laufe dieses Studiums begleitet und auf unterschiedliche Art unterstützt. Ich bedanke mich bei der Gruppe 8 für die Freundschaft, die im Lauf dieser drei Jahre entstanden ist, aber auch bei allen anderen Kolleginnen und Kollegen, mit denen ich zusammen lernen und Zeit verbringen konnte. Des Weiteren möchte ich meine Wertschätzung gegenüber Leon Freudenthaler ausdrücken, der mich während der beiden Bachelorarbeiten sehr gut betreut hat. Mein Dank gilt auch allen anderen Dozentinnen und Dozenten, die sich bemüht haben, Inhalte so zu vermitteln, dass die anfängliche Motivation erhalten bleibt. Besonderer Dank gebührt auch Christine Rauch, Michael Eisl und Nadja Lechner, die freundlicherweise meine Bachelorarbeiten gelesen und korrigiert haben.

Abstract

Authorization and authentication are among the most important security measures, not only but also in the context of microservice-based architectures. Especially the higher number of endpoints compared to the lesser numbers in monolithic architectures requires good security measures to prevent attacks on the system. OAuth2 is a widely used protocol for authorization and OpenID Connect (OIDC) adds an additional protocol layer for authentication to OAuth2. When OAuth2 is implemented in a microservice architecture with an API gateway, there are several ways to distribute the roles defined by OAuth2 among the components involved. This bachelor thesis examines how authentication and authorization can be implemented according to the specifications for OAuth2 and OIDC in a microservice system with Spring Boot and the Keycloak server. The resulting implementation was then checked for compliance with the requirements of OAuth2 and OIDC. Among other things, this revealed that the default JWT Access Token generated by Keycloak does not carry the specific Header that is required according to RFC 9068. Secondly, the difference in performance between two implementation variants is being compared: one where the gateway takes on the role of the OAuth2 client and one where the gateway is implemented as a resource server, with a simulated frontend application as the OAuth2 client. It could be shown that average response times are shorter in the first variant, which suggests that integrating OAuth2 client functionality into the gateway could bring an advantage in performance in comparison to an implementation where the frontend application is also the OAuth2 client.

Kurzfassung

Authorisierung und Authentifizierung gehören zu den wichtigsten Sicherheitsmaßnahmen, nicht nur, aber auch im Kontext von Microservice-basierten Architekturen. Speziell die höhere Anzahl an Endpoints im Vergleich zu monolithischen Architekturen erfordert gute Sicherheitsvorkehrungen um Angriffe auf das System zu verhindern. OAuth2 ist ein vielgenutztes Protokoll für Authorisierung und OpenID Connect (OIDC) ergänzt OAuth2 um eine zusätzliche Protokollschicht für Authentifizierung. Wenn OAuth2 in einer Microservice-Architektur mit einem API-Gateway implementiert wird, gibt es verschiedene Möglichkeiten, die von OAuth2 definierten Rollen auf die beteiligten Komponenten zu verteilen. In der vorliegenden Bachelorarbeit wird untersucht, wie Authentifizierung und Authorisierung entsprechend der Spezifikationen für OAuth2 und OIDC in einem Microservice-System mit Spring Boot und dem Keycloak-Server umgesetzt werden kann. Die resultierende Implementierung wurde anschließend auf die Übereinstimmung mit den Anforderungen von OAuth2 und OIDC überprüft. Dabei stellte sich unter anderem heraus, dass der standardmäßig von Keycloak generierte JWT Access Token nicht den erforderlichen spezifischen Header trägt, der laut RFC 9068 erforderlich ist. Zweitens wurde der Performanceunterschied zwischen zwei Implementierungsvarianten verglichen: bei einer Variante übernimmt das Gateway die Rolle des OAuth2-Clients, bei der anderen ist es als Resource Server implementiert. Dabei konnte gezeigt werden, dass die durchschnittlichen Antwortzeiten bei der ersten Variante kürzer sind als bei der zweiten, was darauf schließen lässt, dass es einen Performancevorteil bringen kann, die Funktionalität des OAuth2-Clients in das Gateway zu integrieren, statt in die Frontend-Anwendung.

List of Abbreviations

ANSI	American National Standards Institute
BCP	Best Current Practice
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
JWT	JSON Web Token
MSA	Microservice Architecture
mTLS	mutual Transport Layer Security
NIST	National Institute of Standards and Technology
OIDC	OpenID Connect
PoC	Proof of Concept
PoLP	Principle of least privilege
RFC	Request for Comments

Key Terms

Authentication

Authorization

BFF

Gateway

JWT

Microservice Architecture

OAuth 2

OpenID Connect

Contents

1 Introduction

In recent years, microservice architectures (MSA) have become increasingly popular. Netflix, Spotify, Amazon, and other well-known corporations have moved to MSA [1], and many others have followed their lead [2] or are planning to do so in the near future [3]. However, besides the well-known advantages such as high flexibility for development, redeployment, and scaling, a small code base in the single services, and low dependency between them, the same features also bring disadvantages that can not be ignored, like the increased attack surface that follows logically from the large number of endpoints which is necessary for the services to communicate between each other [4].

The financial damage caused to companies by data breaches averaged US$ 4.35 million in 2022 [5]. The personal damage that can result for individuals is hardly measurable. Unsurprisingly, not only but also in the context of MSA, topics such as authentication and authorization are therefore high on the list of security issues [6]. In many cases, the damage can at least be limited by not only introducing a general access control but also, for example, distinguishing between different roles so that even a user or system that is legitimate in the organization cannot access areas that do not fall within their scope. One such approach, which is easy to implement, is role-based access control (RBAC) [7].

OAuth 2.0 (OAuth2) is the industry-standard protocol for authorization [8] and also widely used in in MSA [9]. It has been developed for secure third-party access to a resource on a user's or another sytem's behalf, but can also be used to authorize another service of the same system to access a resource, which makes it almost ideal for MSA. OpenID Connect is a layer on top of the OAuth2 protocol which deals with authentication, while OAuth2 only handles authorization.

OAuth2 is defined in RFC 6749 [10] and 6750 [11]. However, there is a number of other relevant documents associated with OAuth2 that have been added over the years. It can be a challenge for developers to keep track, make sure to meet all the requirements for OAuth2 and also take into account current best practices [12]. The long-awaited specification for OAuth 2.1 has been released as Internet-Draft in the attempt to consolidate OAuth 2.0 and its extensions, which should improve this situation [13].

Spring Boot is an open source framework for Java that builds on the Spring Framework. It is widely used and often listed as the top framework for the development of microservices [14], [15], for reasons like fast deployment, embedded servers, dependency management, auto-configuration, code minimization, and extensions that make it easy to build common patterns for distributed systems [16], [17]. One component of the Framework is Spring Security, which offers extensive support for OAuth2. The out-of-the-box auto-configuration of Spring Boot helps to implement OAuth2 with very little boilerplate code, but some manual configuration is still necessary at this level.

Keycloak[1] is an identity and access management (IAM) platform which supports OIDC and RBAC. Like Spring Boot, Keycloak is free to use and open source. During the last years, the Keycloak Adapter for Spring Boot made it easy to integrate Keycloak with Spring Boot applications. However, this adapter is deprecated at the time of writing and not supported anymore by Spring Boot 3. More knowledge of Spring Security is required for the same tasks

[1] https://www.keycloak.org/

[18], [19]. It does not help that the Keycloak documentation is not yet entirely up-to-date at this point and still mentions the use of the adapter. The same goes for most of the tutorials that can be found on the web.

From these initial conditions, the first research question arises: How can authorization and authentication be implemented according to OAuth2 and OIDC, in a MSA with Spring Boot 3 and Keycloak as the authorization server and OIDC provider without the use of the deprecated adapter? For this purpose, the *Teapot*, a small MSA prototype with the functionality of a virtual tea kitchen, has been developed as a proof of concept (PoC), with user authentication and authorization based on roles. The resulting Spring Boot applications and the configuration of the Keycloak server have been analyzed in detail for compliance with the OAuth2 and OIDC specifications with the help of the Wireshark network protocol analyzer[2], the Postman API platform[3] and `jwt.io/` for decoding access tokens and ID tokens (see chapter 3. Developing a fully-fledged RBAC model was considered out of scope, and therefore no analysis for compliance with the RBAC standard was conducted. Instead, roles were configured at the Keycloak server, and their evaluation and resulting access decisions at the microservice level was implemented as a PoC. The implementation was then tested with Postman.

OAuth2 defines among others the role of the resource server, holding the resource to be accessed, the client, and the resource owner, which can be another system or a human user, who wants the client to access a resource on their behalf. In a traditional client-server architecture, there is no question about the interpretation of these roles. The client is usually interpreted as equal to the browser-based or native frontend application, but in a system like MSA, where more components than only one client and one server are involved, the client in the sense of the OAuth2 role does not necessarily need to be the component closest to the resource owner. Instead, the client can also exist within the backend. This comes with several advantages. First of all, the OAuth access tokens are never sent to the browser, and the client can be considered confidential because it is able to hold its own secret client credentials to authenticate with the authorization server [20]. In this thesis, the term *client* always refers to the OAuth2 role. The Teapot prototype was implemented in this manner with both, the resource server, and the client in the backend, leaving a frontend application out of scope. However, since the gateway can be a possible bottle-neck in a distributed system, it is important to consider the performance implications of a gateway that performs the tasks of a client as opposed to those of a resource server. The second research question for this thesis is therefore: What is the impact on the response time of a gateway in the role of the OAuth2 client in comparison to a gateway in the role of a resouce server? To answer this question, this thesis also comprises a performance comparison between the two patterns. For this purpose, the Teapot system was recreated in two simplified versions that differ from each other only in this detail at the gateway level, and a third version without any access control was implemented. To compare the response times of the three versions, load testing was conducted with Apache JMeter[4] (see sections 4.3 and 5.2).

This thesis is structured as follows: the next chapter after this introduction (2) describes other publications which are relevant for this research. Chapter 3 gives an introduction to background and terminology and describes the most important features of MSA, authentication and authorization, RBAC, OAuth2 and OIDC. The implementation of the Teapot MSA prototype, and the setup for load testing with JMeter is described in chapter 4, including a short introduction to Spring (Boot) and Keycloak. Chapter 5 presents the analysis of the

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

resulting implementation and its compliance with the OAuth2/OIDC specifications, as well as the comparison between load testing results of the different gateway roles, and conclusions are drawn in chapter 6.

2 Related Work

Some research papers have been published about securing MSA with OAuth2 and Spring Security. The first to mention is the work of Nguyen & Baker [21]. They present a PoC of an MSA Inventory Management System on which they have conducted security tests to examine possible vulnerabilities. In contrast to the present research, they do not use Keycloak, but implement their own Authorization Server with Spring Security instead. While in the present research the functionality and compliance with specifications is tested by sending valid and invalid tokens with Postman and analyzing Wireshark captures, they conduct Cross-Site Request Forgery (CSRF), Cross-Site Scripting (XXS) and Brute Force attacks to test the efficiency of their implementation. Their architecture does not include an API gateway, the client web application can communicate directly with the resource server. Another difference to the present research is that they do not compare different implementations.

Chatterjee & Prinz [22] created an electronic health coaching (eCoach) prototype system, secured with Spring Security and Keycloak. In their extensive case study they also mention the use of the KeycloakWebSecurityConfigurerAdapter which could not be used in the implementation presented in this thesis because of its deprecation with Spring Boot 3 (see chapter 1). Similar to this thesis, they have also used Wireshark for network traffic analysis and Postman for manual testing of API endpoints. They have also used Apache JMeter, but with the aim to test scalability of their implementation and not for a comparison of different implementations.

Florén [23] has written a Master's thesis about the comparison between different MSA design patterns for authentication and authorization flows, an authorization flow inspired by OAuth2 with JSON Web Tokens (JWT). His work is focused on performance, which has also been tested with Jmeter. It is not clear to which extent the implementation follows the OAuth2 specifications. He found that with increased security (more security checks within the system) come longer response times, but only for high numbers of users. The services themselves were created with Spring Boot, while Express.js was used for the gateways. All compared patterns have in common that authentication and authorization were never implemented in the services themselves, but were always handled by a varying number of gateways, which are the only ones communicating with the authorization server. The present thesis on the other hand follows a different approach, following the statements of Nehme et al. [24], Yarygina & Bagge [25] and others, that no service should blindly trust any other, and that the performance overlay introduced by basic security features at microservice-level is negligible in comparison to the benefits [25].

Unlike the present thesis, none of the authors mentioned above analyses the resulting implementation in detail for compliance with the specifications.

While the more common approach in MSA seems to be that the gateway validates access tokens sent by a client outside of the MSA, thus acting as a resource server, there seems to be little or no academic research about the implications of the MSA gateway in the role of the OAuth2 client instead of the frontend application. However, a similar approach has been described recently as a possible pattern for browser-based or native applications in combination with OAuth2/OIDC by Parecki & Waite [20]. In this Best Current Practice (BCP) Internet-Draft, they use the term *Backend For Frontend (BFF) Proxy* for a server which serves JavaScript code to the browser and is also able to act as a confidential client.

Although the document does not focus specifically on MSA and therefore does not assume that a gateway already exists in the architecture, from this point of view however, the idea to implement OAuth2 client functionality in the MSA gateway does not seem too far-fetched and has also been suggested by developers in blog posts [26], [27]. This is what was implemented for the present research. In contrast to the model described by Parecki & Waite, the gateway in this research does not serve frontend-JavaScript code, and it is left open on purpose whether there is an external frontend or whether it can be loaded by the browser via the gateway itself.

3 Background and Terminology

3.1 Microservice Architecture (MSA)

The first definition for microservices came from Lewis and Fowler in a blog post from 2014 [28]. They describe MSA in comparison to the traditional monolithic server architecture: while a monolith combines all necessary functionality in one piece of software, a MSA is a composition of loosely coupled, separate applications, each with their own domain-logic, running their own process. Figure 3.1 illustrates this difference between MSA and monolithic architecture.

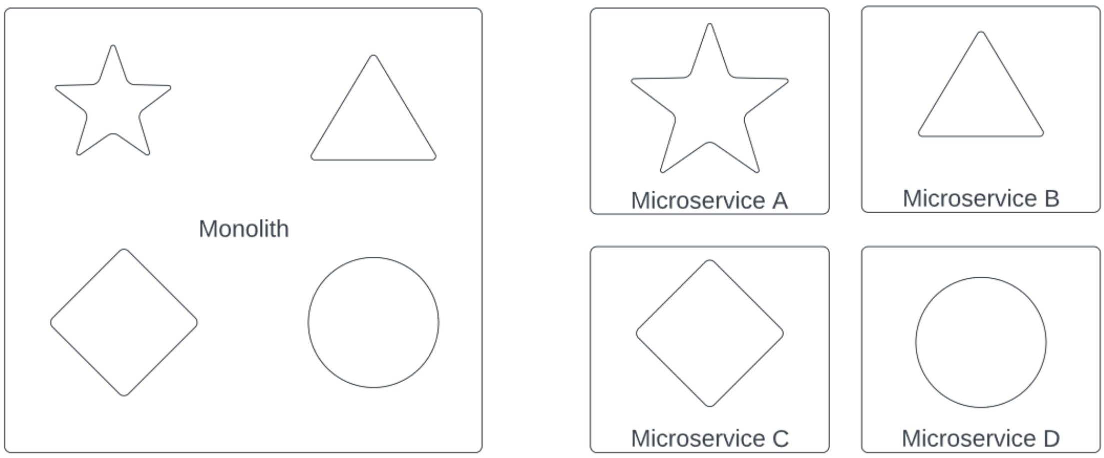

Figure 3.1: Monolithic architecture vs. microservice-based architecture [29] according to [28]

Inside a monolithic server the components communicate within the same process, via method invocation or function call. Microservices on the other hand communicate with each other via a simple protocol, often HTTP with RESTful API endpoints, or leightweight messaging such as RabbitMQ. While there exist more complex structures for the communication between processes, the *smart endpoints and dumb pipes* approach was chosen by the microservice community because it supports best the principle of loosely-coupled services [28]. Of course also monoliths have endpoints and can use RESTful communication, but those endpoints will only be exposed to the outside of the monolith and are not used for internal communication. This difference is important as it has implications for MSA security, as will be explained in section 3.2.3.

The characteristic of microservices being only loosely coupled so important brings several advantages under certain circumstances. These advantages are, according to Dragoni et al. [4]:

- Small code base: The separation into loosely coupled components makes for a small code base within each service, which makes it easier to build and to debug with a shorter down-time.

- Independent maintentance and redeployment of single services.
- Flexible scaling: Smaller parts of the system can be duplicated where and when they are needed, while a monolith can only be duplicated as a whole.
- Flexible development: services can be built using different languages and frameworks, following the circumstances posed by the domain or by the capabilities and preferences of a team, as long as it does not affect the communication between them.

MSA is also very suitable for containerization, and therefore the upcoming of technologies like Docker and Kubernetes has further contributed to the deployment flexibility of microservices [30].

However, MSA is not the best solution in each and every situation. MSA can be a good idea for companies who want to benefit from the scaling flexibility and potentially minimized cost associated to scaling offered by MSA, minimize down-time when availability of all or certain parts of the product is especially crucial for their business, improve development speed by adding more developers, or integrate new technology into their system [31]. However, there are trade-offs between those factors and other circumstances which make MSA not the ideal choice, for example when the domain is unclear or likely to change, when the software product will be installed and maintained by the customer itself and not by the development team or for a small startup that is still experimenting in a new field, before it finds a successful approach [31]. One reason to be cautious when adopting a MSA approach is that MSA poses specific security challenges. These will be addressed in section 3.2.3.

Different architectural patterns of varying complexity have been proposed and studied for MSA [32]. One very common pattern is the API Gateway, which usually lives at the edge of the MSA and functions as entry point for requests coming from a browser, a mobile application or any other kind of frontend [33], so that the single backend services are not required to be reachable from the outside.

3.2 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.

3.2.1 Authentication

Authentication deals with the question of identity. It is often crucial to know the identity of a user or system that sends a request in order to decide if they are allowed to access a specific resource. If this is the case it might also be important to know later, who has accessed the resource, for example if someone has misused their right to access the resource, like in the case of stolen or manipulated data. The process of authentication involves the information who someone is, for example a user ID during a login process 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 [34, pp. 59ff]. A combination of these proves is multifactor authentication which increases the level of security. Not only human persons need authentication, but also systems can possess a kind of ID that identifies them and a prove in the form of a secret code, a token or a certificate. In any case, there must be a database that can be consulted to verify that this prove is valid, otherwise it would be of no value, but it is not necessary for the system that controls access to a resource to possess the database itself, it can delegate the whole business of user authentication to a different entity that functions as an identity provider.

3.2.2 Authorization

Authorization on the other hand is about permissions and access policies, that is, who is allowed to which typ of access to which resources [35]. Although it is a different concept than authentication, authorization is still strongly linked to a subject's identity, and it can not take place without some kind of authentication [34, p. 62], even if it is not always necessary to identify an individual subject. 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 does not 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 or system 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 [34, p. 81]. This too requires the consulting of data to know who is allowed to do what.

3.2.3 Authentication and authorization in the context of 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 [4]. The Open Web Application Security Project (OWASP) lists again broken authentication and broken authorization on top of the Top 10 API Security Risks for 2023 [6]. 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 [4]. While heterogeneity and especially trust might not always be present to the same degree in all MSA systems, the large surface attack area due to the increased number of endpoints and network complexity are inherent features in any MSA system compared to a monolithic architecture.

From the external perspective the experience might not be different when communicating with the MSA system via an API gateway, which hides the underlying complexity of the system. There are only a couple of relevant entry points [36]. 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. Compared to a monolithic system, this makes the implementation and management of authentication and authorization in a MSA more complex.

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 its integrity, regardless of who the sender might be [37], [36]. This approach is important for the mitigation of the confused deputy attack, where a corrupted microservice interacts with other microservices who have no idea that it acts on the attackers behalf [38].

3.2.4 Role-Based Access Control (RBAC)

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 potential damage can be significantly reduced by defining roles with different privileges. RBAC was introduced by Ferraiolo and Kuhn from the National Institute of Standards and Technology (NIST) in 1992 [39] and has later been adopted as a standard by the American National Standards Institute (ANSI) in 2004 and republished several times, last in 2012 [40]. It is an approach to managing access to resources and operations on these resources by distinguishing different kinds of users. Subjects within an organization usually have different functions and qualifications. In RBAC, access permissions are based on these functions. The key elements are subjects, roles, and transactions, and they can be assigned to each other in a many-to-many relationship. Each subject has an active role, but may be authorized to assume other roles as well. Roles determine the transactions that a subject is authorized to perform. Transactions are operations that the subject is allowed to perform, consisting of a type of operation and objects. Subjects are only allowed to perform a transaction if they are assigned a role that is authorized for this transaction. Users are assigned a role that authorizes them to the minimum extent necessary to fulfill their tasks within an organization. For example, not all users need the same access rights as an administrator or manager. RBAC therefore follows the principle of least privilege (PoLP) [39].

3.3 OAuth 2 and OpenID Connect

To begin with, OAuth 2 (OAuth2) is a framework or open protocol for authorization only. OAuth2 is considered the unofficial standard for securing the access to APIs [34, p. 81], [41]. It does not deal with authentication, but with authorization only. It has been described in Request for Comments (RFC) 6749 [10] and RFC 6750 [11], followed by a long tail of further specifications [42], [43], [44], [45], security considerations [46], [47], and a series of Security BCP Internet-drafts [48][1]. The original specifications from 2012 have been updated by RFC 8252, "OAuth 2.0 for Native Apps" [49] and by "OAuth 2.0 for Browser-Based Apps" [20]. Since this is not even a comprehensive list, it is understandable that the new specification for OAuth 2.1 [13], which has been published just recently, has been long awaited by some. It replaces RFC 6749 and 6750.

Because OAuth2 to is not intended to be used for authentication, OpenID connect (OIDC) [50] was created as a separate layer for this purpose, in addition to OAuth2 in 2014. Therefore it should be clear that these two are not alternative concepts to choose from, but when securing services, both should be used.

As already mentioned in chapter 1, OAuth2 requires an authorization server that issues access tokens, OIDC requires an OIDC provider that handles user login. The Keycloak server acts as both and is therefore relevant for the functioning of OAuth2 and OIDC. A description of the Keycloak server and its configuration is given in section 4.2.2.

3.3.1 OAuth2

The motivation for the creation of OAuth2 was to enable a client application to access a resource on a server on behalf of the owner of that resource without the need to pass on the resource owners credentials to this client for authentication [13]. Third-party access to resources is a very common practice, the standard example is an application that needs access

[1] This is not a comprehensive list.

to a user's facebook timeline [34, p. 81], but it may as well be a different resource like pictures or a person's calendar. Sharing a password with a third-party application is not desireable for several reasons: Passwords are inherently weak, especially in combination with human users, who tend to reuse passwords on different unrelated systems, and servers would have to implement support for password authentication [13]. Third-party applications would store these passwords, possibly in clear-text. By authenticating with the server as the user, the third party application would have access to all of the user's data with the same permissions as the user themself. The only way to revoke access for such an application would be to change the password, which would naturally exclude all other third-party applications as well.

OAuth2 separates the *client* (the third-party application) from the role of the *resource owner* (the end user who allows access to a resource to the client), so that the client is not required anymore to pretend to be the user by using the user's credentials when communicating with the *resource server* (the server holding the resource). Instead of a password the client sends an *access token* with every request to the resource server. This access token has a limited lifespan and other attributes that allow to control the extent to which the client can access the resources on that server. This access token is issued to the client by an *authorization server* after the user has given consent to the client to access the resource. Following the principle of separation of concerns, the authorization server is also the only one dealing with authentication of the user [13].

The explicit consent of the resource owner to the client to access the resource is called `authorization grant`. The client can use this grant to obtain the access token from the authorization server. There are several different authentication grant flows defined for OAuth2. The preferred grant type for most use cases is the *Authorization Code* grant type[2]. Other grant types are the *Refresh Token* grant and the *Client Credentials* grant. With the *Authorization Code* grant the authorization server issues first only a code to the user agent together with the redirect url. After the authentication, the user is sent back to the client by the authorization server with this redirect url and the authorization code. The client application can now send the code to a different endpoint at the authorization server, the `token endpoint`, and exchange it for an access token. In this way the access token is transmitted only via backchannel communication, which makes it harder for attackers to intercept the token. There are also additional measurements to make the code exchange more secure, like the `state` parameter and *Proof Key for Code Exchange* (PKCE). When the client redirects the user agent to the authorization server (this is called the authorization request), it must create a code challenge and add it to the request, unless it is a confidential client and the OIDC `nonce` value is used. Other required parameters are the `response_type` (`code` for the `authorization code grant`) and the `client_id` [13]. Figure 3.2 shows the steps involved between the different roles during the authorization code grant flow.

The Access Token

The OAuth2 specification does not define the nature of the access token. Although it can be any arbitrary string that has no further encoded information (this would be a reference token), it is considered best practice to use self-contained tokens like JSON Web Tokens (JWT) because the resource server itself can validate those and determine if the authorization is sufficient for the request, without having to build up a connection to the authorization server

[2]Before OAuth 2.1 there was also the *Implicit* grant and the *Resource Owner Password Credentials* grant, but they are not considered secure anymore. Their use has already been excluded by OAuth 2.0 Securit Best Practice documents[48] and they are completely omitted in the new specification `hardtOAuthAuthorizationFramework2023` (See also [29]).

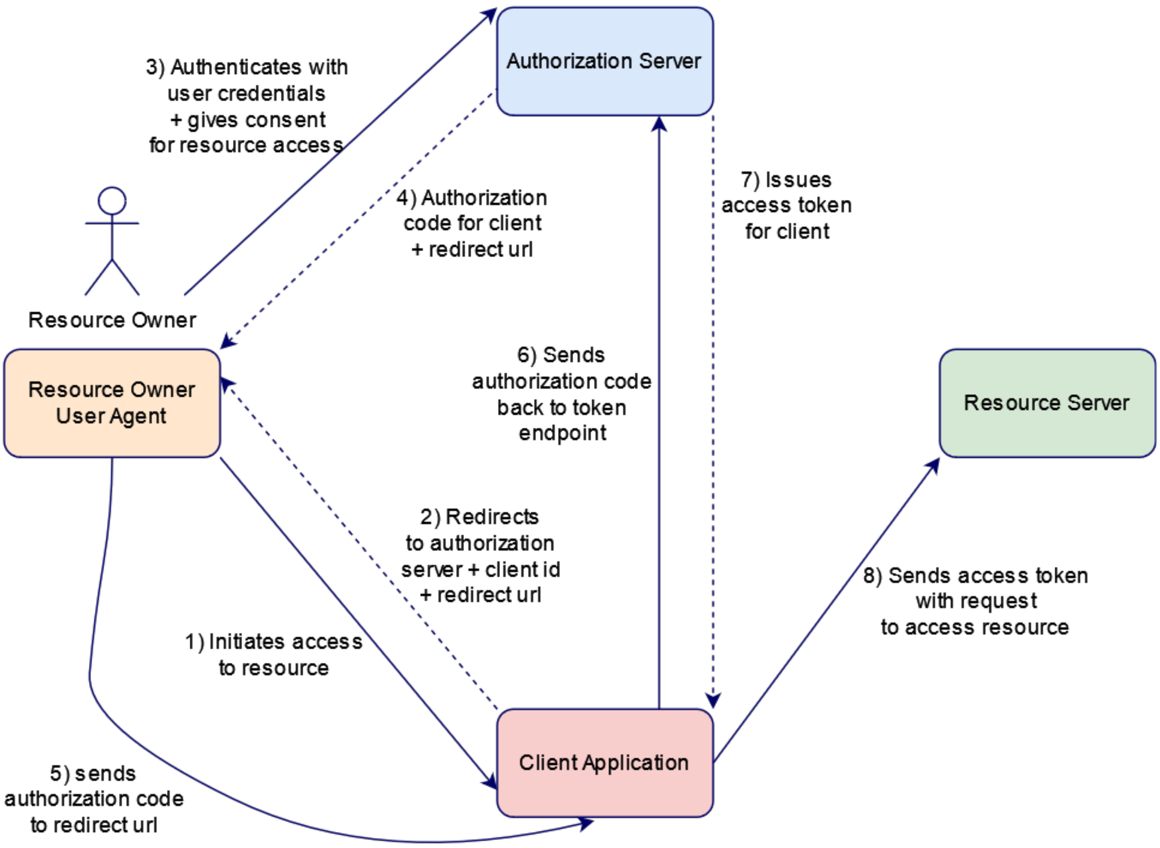

Figure 3.2: Steps of the authorization code grant in the interaction between all roles as described in [13]

```
{
    "access_token":
        "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJHVGZ5eG9HVTBjbGxobzhnX3U1SGcwa
        jNucUtjUjdwMUgzS2xXNi1BQTRBIn0.
        eyJleHAiOjE2ODc5NzUxNTMsImlhdCI6MTY4Nzk3NDk3MywianRpIjoiMGRkYmIxNGQtYjNkNi00MDNmLT
        ljMzYtNWEwYzJlZmMwYzI3IiwiaXNzIjoiaHR0cDovL2hvc3QuZG9ja2VyLmludGVybmFsOjEwMDAxL3Jl
        YWxtcy90ZWFwb3QiLCJzdWIiOiIzMmU1NjQzYy02NWU5LTQ5NDYtOTJjMC04NzdlMWEzNTNlZjciLCJ0eX
        AiOiJCZWFyZXIiLCJhenAiOiJ0ZWFwb3QtZ2F0ZXdheSIsInNlc3Npb25fc3RhdGUiOiI4MDA1NGEyYS1j
        ZWRjLTRlOGMtOTNlMy0zNmI2ZjY1ZDY4NWUiLCJhY3IiOiIxIiwiYWxsb3dlZC1vcmlnaW5zIjpbIioiXS
        wicmVhbG1fYWNjZXNzIjp7InJvbGVzIjpbIm9mZmxpbmVfYWNjZXNzIiwidW1hX2F1dGhvcml6YXRpb24i
        LCJkZWZhdWx0LXJvbGVzLXRlYXBvdCIsInRlYV91c2VyIl19LCJyZXNvdXJjZV9hY2Nlc3MiOnsidGVhcG
        90LWdhdGV3YXkiOnsicm9sZXMiOlsicHJpdmlsZWdlZF91c2VyIiwidXNlciJdfX0sInNjb3BlIjoidGVh
        X3JlYWQiLCJzaWQiOiI4MDA1NGEyYS1jZWRjLTRlOGMtOTNlMy0zNmI2ZjY1ZDY4NWUifQ.
        vbU10Esac38vl7hhoMFGD-CN7JwQGVU6Ff2uzn7ocAgqycfShcidZG3J0X3FFMi91XdC7akc3PGR957ouT
        p1wNUgrjpLz1lQmvIkO6xCPfErh-x6VAX_pXO83hbRqCyKJ5m5fKBEy_dnuhB9vLnOIYRD7YR7eGhHi49z
        IcWLNdxsiCHvjmLHaZOV_jzlO5MgEGjQbsi5ybViKXShl13lLCzU3NNpa6SdnjqydWhQdVkTkX8e-TLQ8x
        TCLvbDKB8FUlJvs4_3dMRZo3HxpqKbqn6ixZZY_P3-ucqV2KZ5HPcti0zX5FWiFuk-mJ3VzPdwO4K1mPIz
        RRal6K0Gxjt2Pw",
    "expires_in": 180,
    "refresh_expires_in": 0,
    "token_type": "Bearer",
    "not-before-policy": 0,
    "session_state": "80054a2a-cedc-4e8c-93e3-36b6f65d685e",
```


Figure 3.3: Token response from the Keycloak server in Postman

each time or maintaining a token database [13]. Specifically, in the context of distributed systems like MSA, the use of JWT as access tokens is advisable. One reason is that they can be validated locally and not every single request to the system has to be validated first with the authorization server, which could result in performance loss [51]. The drawback of local token validation is that if a token has been revoked, the validating service does not know this and will give the client with the token access until it expires [52]. The use of JWTs as access tokens for OAuth2 is specified in RFC 9086 [53].

The client requests access tokens from the `token endpoint` at the authorization server with a POST request including at least the `client_id` and the `grant_type` [13]. Confidential clients, which are capable of maintaining secret information, will also include a form of authentication, either symmetric (password) or asymmetric (a signed JWT or mutual Transport Layer Security (mTLS)). It is also possible for the client to include a `scope` request parameter. This is necessary for example when using OIDC (see section 3.3.2).

The response from the authorization server, if the authorization request was valid, includes several parameters [13]: the freshly issued access token for the client and `token_type`, which is usually `"Bearer"`, as well as a `scope` parameter are required by the specification. The authorization server can decide if the scope specified in the request will be included. Further recommended is `expires_in`. A refresh token can also be included in the response, depending on the type of client and other factors. Figure 3.3 shows a token response from Keycloak. The lifespan for the access token is configured to be three minutes and because the authenticated user has only *read* privilege, the scope `tea_read` is included.

A JWT is base64-encoded and consists of three parts, the header, payload and the signature. Any JWT that is used as access token for OAuth2 must be signed, as specified in RFC 9068, to ensure its integrity, preferably with asymmetric cryptography [53]. The header contains the `alg` parameter indicating the signing algorithm, which has to include `RS256`, and the `typ` parameter, which has to be `at+jwt` (recommended) or `application/at+jwt`.

The payload contains the claims set where the following claims are mandatory for Oauth2 access tokens [53]:

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

An example for a decoded access token is shown in figure 3.4, although it does not conform entirely to the specifications, as the `typ` header parameter contains `JWT` instead of the required `at+jwt` value. The client ID is instead present with the `azp` claim.

When validating the access token, the resource server checks the signature, lifespan, scope and possibly other authorization parameters for the specific resource [13]. The signing keys can be requested by the resource server from the authorization server in a JSON Web Key Set (JWKS), which the authorization server can offer as a part of its metadata the well-known metadata endpoint [54].

3.3.2 OpenID Connect (OIDC)

When it comes to authentication of end-users, OIDC [50] gives the answers that were left out by OAuth2. The content of an OAuth2 access token is of no interest to the client. When the client needs information about a resource owner's identity, it can request an additional *ID token* by adding `openid` to the `scope` parameter in the authorization request, which thus becomes a authentication request. Also the `redirect_uri` parameter is now required. Among other optional parameters, the `nonce` parameter is worth mentioning. It is a string value, used to protect against replay attacks, that should be unfeasible to guess and it represents session state between the client session and the ID token. The ID token will then be issued by the authorization server which is now called OIDC provider, together with the access token and delivered as a part of the token response [50]. The ID token itself is a signed JWT which contains the `iss, sub, aud, exp` and `iat` claims as a requirement. When using the authorization code flow, also the `at_hash` claim is required. It contians the base64 encoded left most half of the corresponding access token's hash. The `nonce` claim is required if it was present in the authentication request. It must have the same value, which will be verified by the client. Other optional claims are `amr` (authentication method references), `acr` (authentication context class reference) and `azp` (authorized party), containing the client ID of the authorized client. The ID token itself does not contain personal user information. Instead there is a separate OAuth2 protected `/userinfo` endpoint where the client can request additional metadata about the user with the access token [50]. An example for an ID token is shown in figure 5.1 in chapter 5.

HEADER: ALGORITHM & TOKEN TYPE


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


PAYLOAD: DATA


```
{
  "exp": 1687724956,
  "iat": 1687688956,
  "jti": "b638eff6-bc73-4eb8-9900-27818d303e02",
  "iss": "http://host.docker.internal:10001/realms
/teapot",
  "sub": "f6ed8914-a2dd-4800-aaf2-9762c835006d",
  "typ": "Bearer",
  "azp": "teapot-gateway",
  "session_state": "86243327-51b4-4eee-9b37-
bccf10cc008e",
  "allowed-origins": [
    "*"
  ],
  "realm_access": {
    "roles": [
      "offline_access",
      "uma_authorization",
      "default-roles-teapot",
      "tea_user"
    ]
  },
  "resource_access": {
    "teapot-gateway": {
      "roles": [
        "privileged_user",
        "user"
      ]
    }
  },
  "scope": "profile email",
  "sid": "86243327-51b4-4eee-9b37-bccf10cc008e",
  "email_verified": false,
  "preferred_username": "user1",
  "given_name": "",
  "family_name": ""
}
```


Figure 3.4: Example of an access token issued by the Keycloak server.

3.4 The positioning of the OAuth2 client in a MSA system

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. The most simple implementation would probably be to implement a 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 mTLS, supposedly the most popular option (see [55, 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 requesting entity 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 [56]. 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 [55, pp. 161ff]. A simplified version of this concept has been implemented and tested for this thesis (see chapter 4). In both cases however, the job position of a (registered) OAuth2 client is still vacant. There should be an application that is able to determine if the requesting entity has already been authenticated, refer it to the OIDC provider for authentication and obtain the access token on the requesting entity's behalf. If the gateway at the edge of the MSA is a resource server, this means that any possible frontend has to implement OIDC client functionality. This comes with additional security challenges. Browser-based applications and native applications are so-called public clients, which means that they are not able to hold secret credentials. With the OAuth2 security BCP [48], the authorization code grant with PKCE is now required for these types of clients, but even this does not mitigate all the risks that come with a public client. For example, access tokens and refresh tokens are still sent to the user agent with browser-based apps [57].

A different version is the BFF Proxy, which has already been mentioned in chapter 2. It implements the gateway directly with the functionality of the OAuth2 client, so that the frontend application will never see an access token or a refresh token. A BFF Proxy also serves the frontend JavaScript code to the browser. The gateway stores access tokens and refresh tokens and keeps a cookie session with the browser-based app. The advantage is that attackers can not steal tokens from the frontend. However, the risk remains, that attackers may send authenticated requests to the gateway impersonating a legitimate user. Therefore additional protection of the session cookies is necessary [20].

For the present research, a similar approach has been chosen: the client functionality was implemented in the gateway, but the frontend application was left out of scope. It could be served by the gateway, like in the BFF Proxy, but also in a separate webserver or in a native application. This is also the reason why the term BFF is not used in this thesis. According to Newman [58], authentication or authorization functionality alone is not sufficient to consider the gateway a BFF. The implementation is described in detail in section 4. The services beyond the gateway are resource servers, so that the communication between the API gateway and the next downstream service is secured via access tokens.

4 Implementation of the Teapot MSA application

The *Teapot* is a virtual tea kitchen with the purpose to serve as an experimental prototype for the implementation of OAuth2/OIDC with Spring Boot and Keycloak for the first research question presented in chapter 1. It represents the backend system to a possible frontend application which could consume the gateway's endpoints, but the frontend itself was left out because it is outside the scope of this thesis. Without a frontend application it was a logical choice to assign the gateway the role of the client which obtains access tokens on behalf of the user. The gateway therefore is not a third-party application in the organizational sense, but belongs to the same organizational entity (the Teapot) as the resource. The services behind the gateway are resource servers because they hold the resource that is requested via the gateway. The following sections explain the implementation in detail and chapter 5 presents an analysis of this implementation and its compliance with the specifications for Oauth2 and OIDC.

In a later stage of this research, roles were created at the Keycloak server and assigned to registered users. Among different possible solutions, the choice was made to implement the check for user roles at the resource server level. This solution is intended to serve as a minimalistic PoC for the consideration of different roles during the access decision process by extracting the role from a user's access token.

In order to compare the perfomance impact of the client position at the gateway or outside the MSA at frontend-level for the second research question presented in chapter 1, the Teapot system was rebuilt in a more minimalistic way, reducing unnecessary overhead that would only blur the result. Three versions were built from this minimalist Teapot system: One with the gateway as client, as in the first version mentioned above, one with the gateway as a (second) resource server, assuming that a hypothetical front end would be the client, and one without security implementations, to have a reference to compare the performance overhead.

4.1 The Teapot - High-level design

In the Teapot system the user can view a list of available types of tea and request a "cup" of the chosen tea. The MSA 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 end user. It routes requests to the Tea service and to the Milk service respectively, so that the user or any front end does not have to communicate directly to the services beyond the gateway. A Keycloak server is deployed for user authentication and issuing of access tokens, serving as OIDC provider and authorization server. The high-level architecture of the teapot is depicted in figure 4.1

In this implementation, the gateway is a OAuth2 client and the Tea service and the Milk

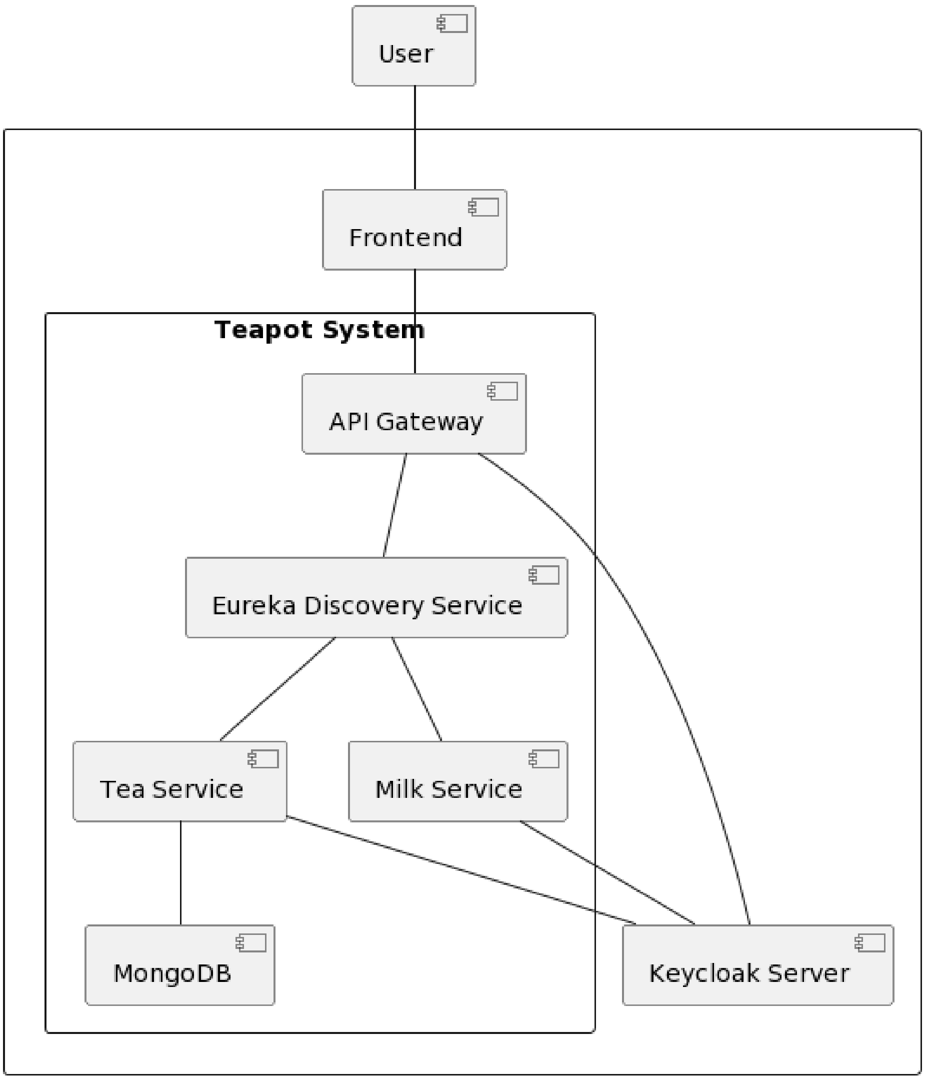

Figure 4.1: High level diagram of the implemented services and their relation to each other

service are resource servers. The first request to a protected resource, when the gateway has not obtained an access token yet, can be depicted as in figure 4.2. This is a simplified diagram, leaving out the discovery service and the database.

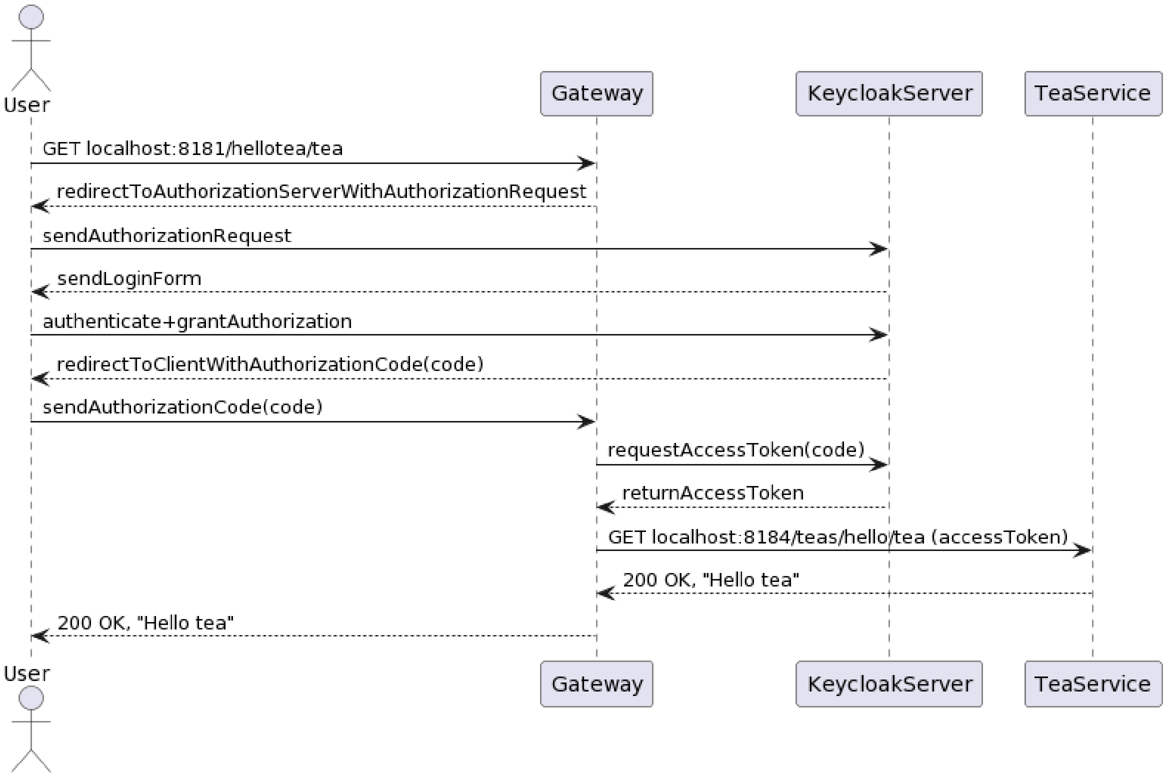

Figure 4.2: Sequence diagram of the first request to a protected resource including a simplified auth code grant flow

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 check the session cookie and, if the user is already authenticated, append the access token to the request as authorization header and forward it to the Tea service.

The resource server must obtain the JSON Web Key Set (JWKS) from the Keycloak Server, so that it will be able to validate the signature of the access token. This happens at the first request.

4.2 Setup with Spring Boot and Keycloak

4.2.1 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 [59]. 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 [60, pp. 4f]. This helps to reduce the need for the developer to write

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

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, as is the case for security configurations [60, p. 50], either programmatically with Java or in many cases by adding configurations to the `applications.properties/applications.yml` file [61]. 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.

4.2.2 The 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, the default database (dev-file) was sufficient for the Teapot project. The Keycloak server already contains a `master realm` with the administrator account. The `master realm` is the parent of all other realms that can be created by an administrator. For this project, a `teapot realm` was created. Inside a realm, administrators can create (register) and manage clients and users. The Teapot gateway needs to be registered as a client in the Teapot realm. When it is created, the client secret is set automatically for the new client, if `Client authentication` is enabled. This is possible because the Teapot gateway is a confidential client. The secret is used by the gateway application when connecting to the Keycloak server to authenticate itself.

For the Teapot project setup where all services are deployed with Docker Compose, the `Frontend-Url` is set to `http://host.docker.internal:10001` where the host name is the docker network and the port is the port assigned to Keycloak. If the `Frontend-Url` is not set explicitly, the host name for the Keycloak endpoints that are used for authentication and authorization flows is set to `localhost`. Services in other Docker containers access the Keycloak server under its `host.docker.internal` url. The frontend-url also determines how the `iss` claim is set in access tokens, which must be identical with the `issuer-uri` set at the resource server [62]. 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 4.3).

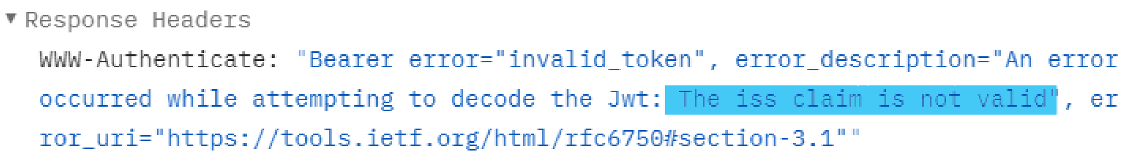

Figure 4.3: Response header indicating that the `iss` claim is not valid

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

All endpoints of the Keycloak server for a realm are accessible under `<host:port>/realms/<realmname>/.well-known/openid-configuration`. 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 4.4 shows the first part of these endpoints.

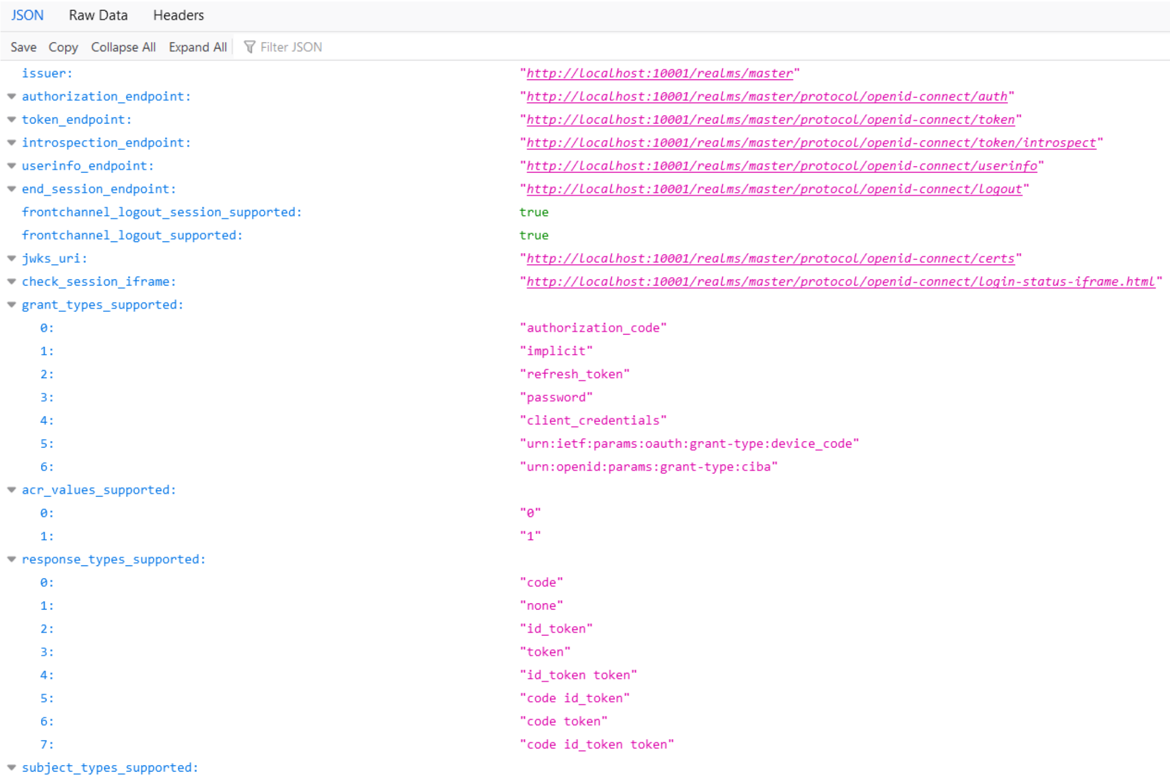

Figure 4.4: Keycloak endpoint configuration for the teapot realm (not complete)

4.2.3 The Spring Cloud Gateway as OAuth2 Client

The gateway's job in a MSA is to route requests from the edge to services inside the MSA (see also sections 3.1 and 3.4. 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>
```

Listing 4.1: Spring Cloud Gateway starter dependency

With the Spring Cloud Gateway implemented, a Handler Mapping checks incoming requests for matches with configured routes and if so, forwards them to the Gateway Web Handler. The request then goes through a filterchain where route-specific pre- and post-logic is applied. Routes can be configured in the `application.properties` file or in the `application.yml` [63].

In order to configure the gateway as OAuth2 client, the `spring-boot-starter-oauth2-client` dependency needs to be included in the `pom.xml` file:

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

Listing 4.2: Spring Bott Oauth2 Client starter dependency

Here it is important to choose the correct starter dependency and not to 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 [64]. Then, after having created the client in Keycloak (see section 4.2.2), 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 4.3).

```
spring:
[...]
  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 4.3: OAuth2 client configuration in the gateway's application.yml file

For this purpose the `spring.security.oauth2.client.registration` base property prefix is used, 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 3.3.2, `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 3.3.1). 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 will not 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 [65].

The gateway must also be able to attach access tokens to any authorized request that will be routed to a downstream resource server (see section 3.3). Spring Security offers a `TokenRelayGatewayFilterFactory` which retrieves the access token that is stored for 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 4.7. This filter will then be applied to all configured routes. Alternatively, the filter can be configured for specific routes by adding `- TokenRelay=` to `filters` [63].

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

        [...]

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

      default-filters:
        - TokenRelay=
```

Listing 4.4: Route configuration with token relay default filter in the gateway's application.yml file

Security configuration for the gateway's endpoints can be added in the way that is shown in listing 4.5. Because `/hellogateway` and `/hellotea/noauth` should remain open for testing purposes, this is taken care of 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 [66].

```
@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 4.5: SecurityWebFilterChain configuration for the OAuth2 client

One particular aspect here is the `logoutSuccessHandler` call that gets a `ServerLogoutSuccessHandler` object as an argument. A separate bean, as shown in listing 4.6, 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 [67], [68], [69]. This process is defined in OpenID Connect Session Management 1.0 as the *RP-Initiated Logout*, where RP stands for relying party [70]. 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 [69]. Figures 4.5 and 4.6 show the process in the Firefox networks analytics tool. First, a POST request is sent to the Teapot gateway's `logout` endpoint, then a redirect follows to `http://host.docker.internal:10001/realms/teapot/protocol/openid-connect/logout`, which is the `end_session_endpoint` at the Keycloak, together with the `id_token_hint` and the `post_logout_redirect_uri` as query parameters. The `id_token_hint` is used to let Keycloak know for which user the session should be cancelled. The `post_logout_redirect_uri` is open to anonymous users and does not require authorization.

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

Listing 4.6: . Code example from the Teapot gateway according to [69]

Status	Method	Domain	File	Initiator	Type
302	POST	localhost:8181	logout	document	html
302	GET	host.docker.internal:10001	logout?id_token_hint=eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJHVGZ5eG9	document	html

Figure 4.5: POST request to the `logout` endoint of the Teapot gateway and redirection to Keycloak's `end_session_endpoint`

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

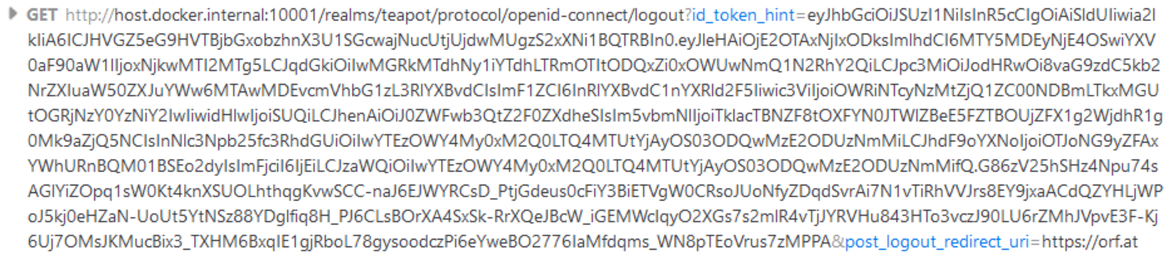

Figure 4.6: Request to the `end_session_endpoint` with query parameters

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 authentication 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 4.7: Reading the user's name from the authentication principal

As already mentioned in chapter 3, the gateway as a client keeps a session with the frontend application. Session management works with Spring Boot out of the box by default without explicit configuration [71]. The next section also shows how a connection is configured to be stateless.

4.2.4 The Resource Server

The OAuth2 Resource Server receives and validates the access token and, if the token is valid, grants access to the requested resource (see section 3.3.1). 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 [72].

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 [62]. Both are included in the `spring-boot-starter-oauth2-resource-server` starter dependency. OAuth2 bearer token authentication is possible with JWTs or with opaque tokens. The Teapot project works with JWT as it is considered best practice (see section 3.3.1).

The authorization process when a request for a protected resource comes in without an access token, goes like this [73]:

- 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 figures 4.7 and 4.8)
- Now the client can retry the request with the bearer token.

```
public final class BearerTokenAuthenticationEntryPoint implements AuthenticationEntryPoint {
    no usages
    private String realmName;

    no usages
    public BearerTokenAuthenticationEntryPoint() {}

    no usages
    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; }
```

Figure 4.7: The BearerTokenAuthenticationEntryPoint sends a `WWW-Authenticate: Bearer` header back to the requesting client [74]

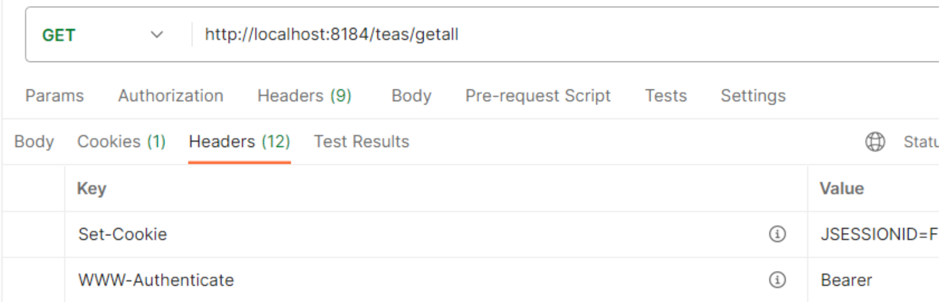

Figure 4.8: `WWW-Authenticate` header in the response to an unauthorized request to the resource server

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 [75] . The `Authentication` represents the authenticated user and contains (among others) a `principal`, which represents an individual, corporation, login ID or any other entity [76], crentials and authorities [77]. An `authority` is an instance of `GrantedAuthority` and usually represents coarse-grained permission, for example `role` or `scope` [78] (for more details about the implementation of role-based access control, see section 4.2.5). The `credentials` in this case contain the access token. The `Authentication` 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 [73].

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 query the jwks-url for the public key set of these algorithms in the next step. Lastly, the validation strategy will be configured to check the iss claim of received JWT access tokens against the given issuer-uri. For this reason the authorization server must be reachable, or else the resource server application will fail at startup [62].

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 does not need to call the issuer-uri in order to know the end point to retrieve the JWKS [72]. In the Teapot project's application.properties file, this looks like in listing 4.8. Still, with the issuer-uri provided, the iss claim in incoming JWTs will be validated against the given issuer [62].

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 [62]. 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 4.2.5).

```
spring:

 [...]

  security:
    oauth2:
      resource-server:
        jwt:
          issuer-uri: ${KEYCLOAK}/realms/teapot
          jwk-set-uri: ${KEYCLOAK}/realms/teapot/protocol/openid-connect/certs
```

Listing 4.8: issuer-uri and jwk-set-uri in the Tea service's application.properties file

Like with the client (see section 4.2.3), the SecurityFilterChain bean can be overridden for custom configuration. A minimal configuration of the SecurityFilterChain can look like in listing 4.9. 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 [79].

```
@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 4.9: `SecurityFilterChain` configuration in the Tea service (resource server)

For the definition of authorities, which 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(Converter)` instead of `OAuth2ResourceServerConfigurer::jwt` [79].

Spring Security has CSRF protection enabled by default [80]. Because the resource server relies on bearer token authentication, some authors in grey literature recommend making the session stateless [19] and, as a consequence, to disable CSRF protection [81]. This is only possible for resource servers , while clients that are consumed by browsers and rely on session cookies must always enable CSRF protection [48]. Making the session stateless can be done in the `SecurityFilterChain` bean as shown in listing 4.10.

```
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        // [...]

        http.sessionManagement((session) -> session.sessionCreationPolicy(
    SessionCreationPolicy.STATELESS))
                .csrf().disable();
        return http.build();
    }
```

Listing 4.10: 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 which 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.

4.2.5 Role-based access control with Keycloak and Spring Boot Resource Server

As an extension to the answer to the first research question, different user roles are defined with Keycloak and checked by the resource server before responding to the request. This is a very minimalistic interpretation of RBAC (see section 3.2.4). In the example of the Teapot this means that someone with a basic user role is not authorized to perform any operation on the Tea database that is not safe, like creating or deleting tea. 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.

With Keycloak there is the option to define realm roles and/or client roles. Realm roles can be composite roles by being associated to other realm roles or client roles [82]. Figure 4.9 shows the defined client roles for the Teapot gateway , while figure 4.10 depicts one of the composite realm roles, `tea_admin` and its associated client roles, `admin`, `user`, and `privileged_user`, which belong to the `teapot-gateway` client.

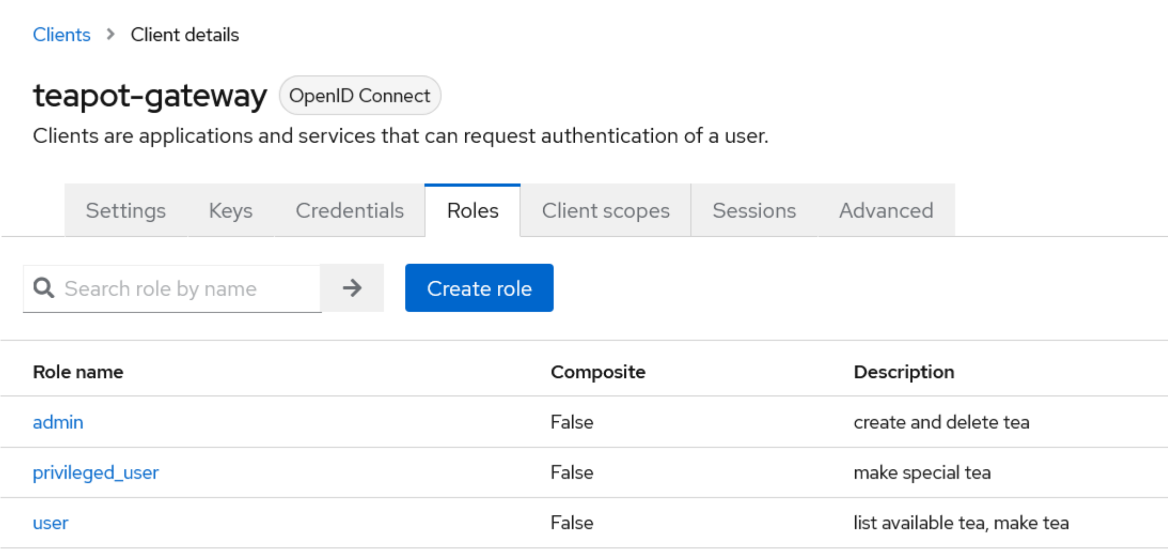

Figure 4.9: Example of client roles defined in the Keycloak admin console

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 3.4. 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 issued by Keycloak does not contain `authority` claims, but `realm_access` or `resource_access`. Spring Security allows to check for authorities inside the `authentication` object, but it provides no means by default to check specifically for the access claims provided by a Keycloak access token. This means that these roles have no effect on the authorization process, unless an authority converter is added. A converter translates specific claims in the access token to an `authority` in order to distinguish roles in authorization. Listing 4.11 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 [83]. This prefix has to be added in the conversion process as well, as shown in listing 4.11, 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>` [62]. Listing 4.12 shows the custom converter that is

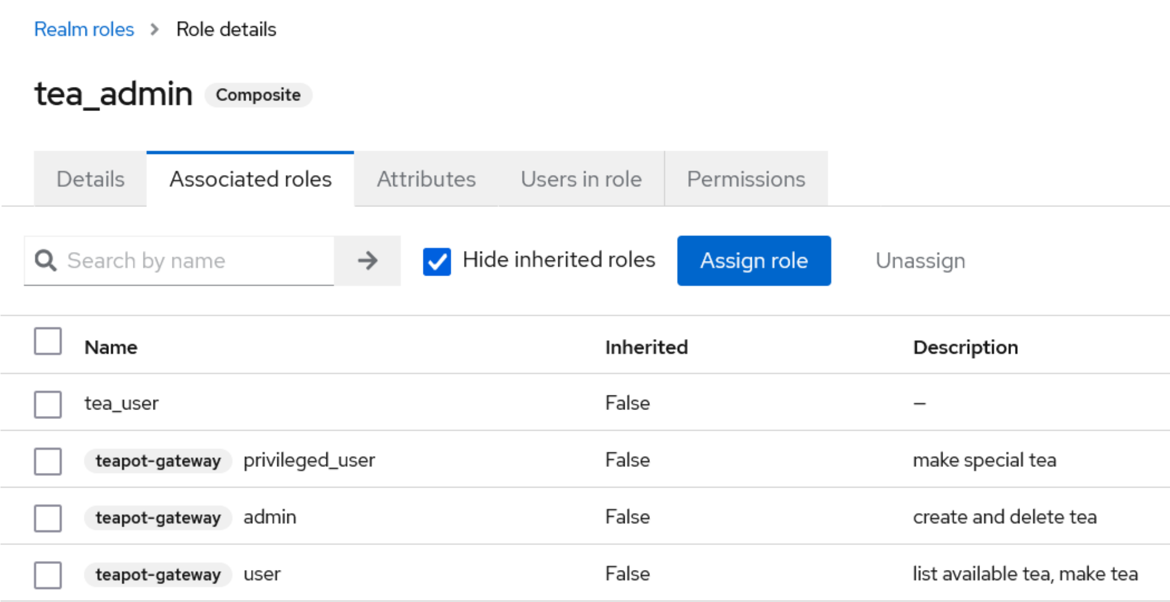

Figure 4.10: Example of a composite realm role and its associated client role in the Keycloak admin console

used in the Tea resource server. It returns a `JwtAuthenticationToken` which inherits from `AbstractOAuth2TokenAuthenticationToken` and contains the extracted realm roles and client roles from the access token as authorities. As it turns out, with the converter code proposed by [84] it is important that the profile scope is included in the access token. Keycloak will then add further profile information, including the `preferred_username` path, to the token. As an alternative, the try-catch block was added in the present implementation so that the `Authentication` is created only with the value of the `sub` claim.

```
@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 4.11: Extraction of client roles and realm roles from Keycloak access token and conversion to granted authorities according to [84] - simplified with added `ROLE_` prefix

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

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

Listing 4.12: Custom converter to set the extracted granted authorities from the access token in the new `Authentication` according to [84] with added try/catch blocks

The `SecurityFilterChain` bean can now be overridden in a way so to grant access to different endpoints of the service, or not, depending on the role authority extracted from the access token of the user who requests a resource. The custom jwt converter from listing 4.12 is given as parameter to the `oauth2resourceServer` overload method. In listing 4.13, 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` therefore has 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 listing 4.13).

```
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
```

```java
    public SecurityFilterChain filterChain(HttpSecurity http, Converter<Jwt, ? extends AbstractAuthenticationToken> jwtAuthenticationConverter) throws Exception {
        http
                .authorizeHttpRequests().requestMatchers("/teas/hello/noauth")
                .permitAll()
                .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();
    }
}
```

Listing 4.13: Custom SecurityFilterChain with role-mapping for protected endpoints

When the Tea service now receives an access token from a user with the wrong role, it responds with the `403 Forbidden` code and a `WWW-Authenticate` header with the message that higher privileges are required (see figure 4.11).

```
▾ Response Headers
  WWW-Authenticate: "Bearer error="insufficient_scope", error_description="The request requires higher privileges than provided by the access token.", error_uri="https://tools.ietf.org/html/rfc6750#section-3.1""
```

Figure 4.11: `WWW-Authenticate` response header for a request with unsufficient priviledges

4.3 Load testing with JMeter

This section describes the implementation and test setup for the answer to the second research question presented in chapter 1.

Since there is a number of functionality present in the previously presented implementation that is not necessary for the comparison of the two client positions, the whole system was rebuilt in an even simpler version for the second part of the research. It was reduced to the gateway, one additional service for the gateway to communicate with, and also contains the keycloak server. The gateway and the Tea service offer "hello"-endpoints that were used for debugging in the beginning. These endpoints were used for load testing, so the database was not involved.

The remaining stripped-down system is represented by figure 4.12.

As already mentioned at the beginning of this chapter, this version was built three times: with the client at the gateway as described in section 4.1, with the client as resource server and a fictional client at the frontend level, and without any security implementation. For the

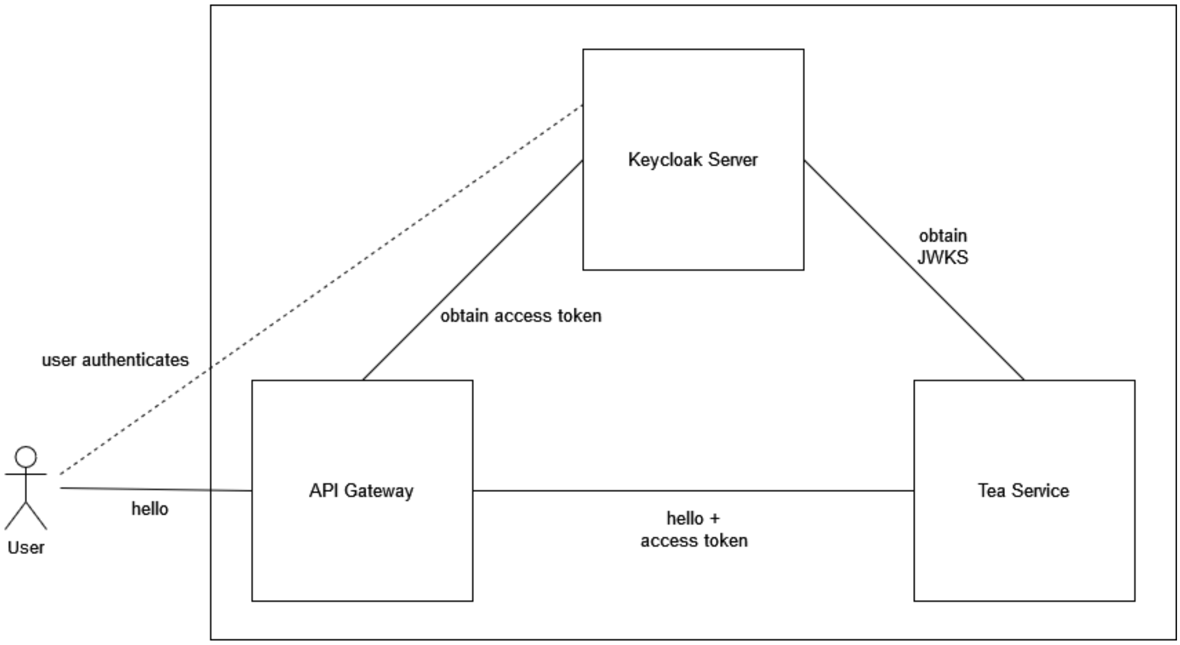

Figure 4.12: High level diagram of the implemented services and their relation to each other

load testing, JMeter imitated the client as it had an access token that it could send to the gateway (as resource server).

For testing purposes, an endpoint at the respective gateway was used that forwards the request and the answer to and from the Tea service. When the gateway is a client, it has to ckeck the session cookie, find the stored access token, and append it to the request before routing it to the Tea service. When the gateway is a resource server, it has to validate the received access token. In both scenarios, the Tea service is a resource server as described in section 4.2.4. For authentication in the first case, user login was first triggered via the browser so that the client would send a session cookie to the browser and store the user's access token. This session cookie was extracted and sent to the client-gateway with each request. In the second case, the gateway as resource server expects an access token, so this token was generated with Postman and sent with each request.

The load testing was performed with Apache JMeter and the services were deployed with Docker Compose[4] on the same machine. All testing was carried out on a Lenovo Thinkpad T490s with a Intel®Core™i78665U CPU, 1.9 GHz Base Frequency, 4,8 GHz Max Turbo Frequency and 4 Cores. The operating system was Microsoft Windows 11 Pro.

Testing parameters were the number of threads (which simulate the number of users), a *ramp-up period* of 0, which means that all threads were started together and a *loop count* of 1, so that each thread will send the request only once [85]. The tests were performed in several rounds for 100 up to 500 concurrent users where 50 users were added to each round. For higher numbers of users a part of the requests started to fail.

For a comparison, the average response times were collected from the HTML dashboard generated by JMeter for each testround. Load testing results are presented and discussed in section 5.2.

[4]https://docs.docker.com/compose/

5 Analysis and Discussion

The following sections refer to the research questions by analyzing the network traffic generated by the implemented solution and by presenting the load testing results.

5.1 Compliance with OAuth2 and OIDC specifications

For the implementation presented in chapter 4, functionality and compliance to OAuth2 and OIDC specifications were tested with Postman and Wireshark.

The network traffic from a request to access the protected resource (the list of available Teas) via the gateway in its client function at `http:localhost:8181/listteas` and with Firefox browser as user agent was captured with Wireshark and is listed in appendix 6.

It can be shown that the steps of the authorization code flow match with the OAuth2 specification. The authorization code is passed through via the user agent to the client which then trades it for the access token with the authorization server. Authorization request, token request, and token response are present and the access token is passed on from the client to the resource server as bearer token in the `Authentication` header.

The authorization/authentication request also contains the required parameters `response_type`, `client_id`, `redirect_uri` and `scope` with `openid` as the value [13], the last two parameters being required by OIDC [50]. The `code_challenge` parameter is missing. However, it is not required when the optional `nonce` parameter is used, which is the case here, but still recommended by the OAuth2 specification [13]. Also, the recommended `state` parameter is present.

The token request is sent to the Keycloak server's token endpoint and again contains the `client_id` and `grant_type`, as required. The client authenticates during the token request with basic authentication. This is possible by definition of OAuth2[13] and it was a deliberate choice during the implementation. However, Spring Security and Keycloak both support other methods of client authentication, for example with a signed JWT [86], [87]. The value in the authorization header is the base64-encoding of `client_id:client_secret`.

The token response also contains all required parameters, including the ID token, since it is an OIDC token response. The `scope` parameter contains more scope strings than the requested `oidc` scope and is therefore required as well [13]. Other parameters are specific to Keycloak and are not covered in the specifications. The access token life time was intentionally configured to be longer than recommended for testing convenience, but can be configured to any shorter period in seconds.

The access token itself turned out to show some flaws. In the Wireshark capture it is truncated and can not be anaylzed. However, when examining other access tokens issued by Keycloak, some faults become apparent: The media type in the `typ` header parameter is not `at+jwt` or `application/at+jwt` as required for OAuth2 JWT tokens. This special value is important for the distinction between access tokens and ID tokens and to avoid that ID tokens are accepted as access tokens [53]. In fact, the attempt to request Tea from the Tea service with the ID token instead of an access token was successful with this implementation (without role mapping). No configuration option could be found for Keycloak to change the token header, neither in the documentation nor by browsing through Keycloaks admin console. This does not mean that it is not possible, but it can be criticized that it is not as straightforward as it should be and the question remains why the required media type is not the default. As long as the access token's media type can not be changed, implementing its validation on the resource server would not make sense. A related problem is the missing `aud` claim in the access token, which must be validated by the resource server that must reject the request if it can not identify itself with the value [53]. The `aud` claim can be set and configured in the Keycloak admin console by adding a scope mapper of the *Audience* type. Token validation with Spring Security can be configured with the `OAuth2TokenValidator` API [62]. At the Tea service, this would look like shown in listing 5.1. An example access token which now includes the required `aud` claim is shown in figure 3.4. Another required parameter not included by Keycloak by default is `client_id` [13]. Adding it to the token with a scope mapper can be done in a similar way as for the `aud` claim, although Keycloak does not seem to offer a preconfigured mapper for this purpose. However, the `azp` claim does indeed contain the client ID by default and seems to fulfill a similar purpose.

```
OAuth2TokenValidator<Jwt> audienceValidator() {
    return new JwtClaimValidator<List<String>>(AUD, aud -> aud.contains("tea"));
}
```

Listing 5.1: Validator for the `aud` claim for the Tea resource server according to [62]

The ID token issued by Keycloak is inconspicuous in comparison. An example ID token with additional user information is shown in figure 5.1. Including user profile information or roles in the ID token is optional and can be configured in the Keycloak admin console. The same is true for the information returned by the userInfo endpoint.

5.2 Response times

Load testing of the different gateway variants showed a tendency toward higher average response times at the gateway as resource server in comparison to the gateway as client. The response times for the unprotected gateway were generally lower than the other two gateways, which can be expected because no endpoint protection mechanism is involved in the process. Figure 5.2 shows the average response times for all three variants.

The load testing experiment was repeated several times. Although the resulting numbers were not always the same as in the graph depicted in figure 5.2, they consistently showed that the gateway has longer response times when it is implemented as resource server than in its client implementation. The difference betwen the two gateways is that the gateway as resource server receives the access token, checks its validity and routes it forwards with the request, while the client-gateway receives a session cookie that will be matched with the one stored for the user agent. It then takes the access token that it associates with the authenticated user and routes it with the user's request to the resource server. It does not have to read the access token for that purpose. The effect might even be stronger with a more thorough examination of the access token when more parameters have to be checked before the token is accepted. Additionally, when the gateway is a resource server, a frontend application that consumes the gateway's endpoints has to implement client functionality which includes occasional communication with the authorization server. This can only add to the total delay from the user's perspective.

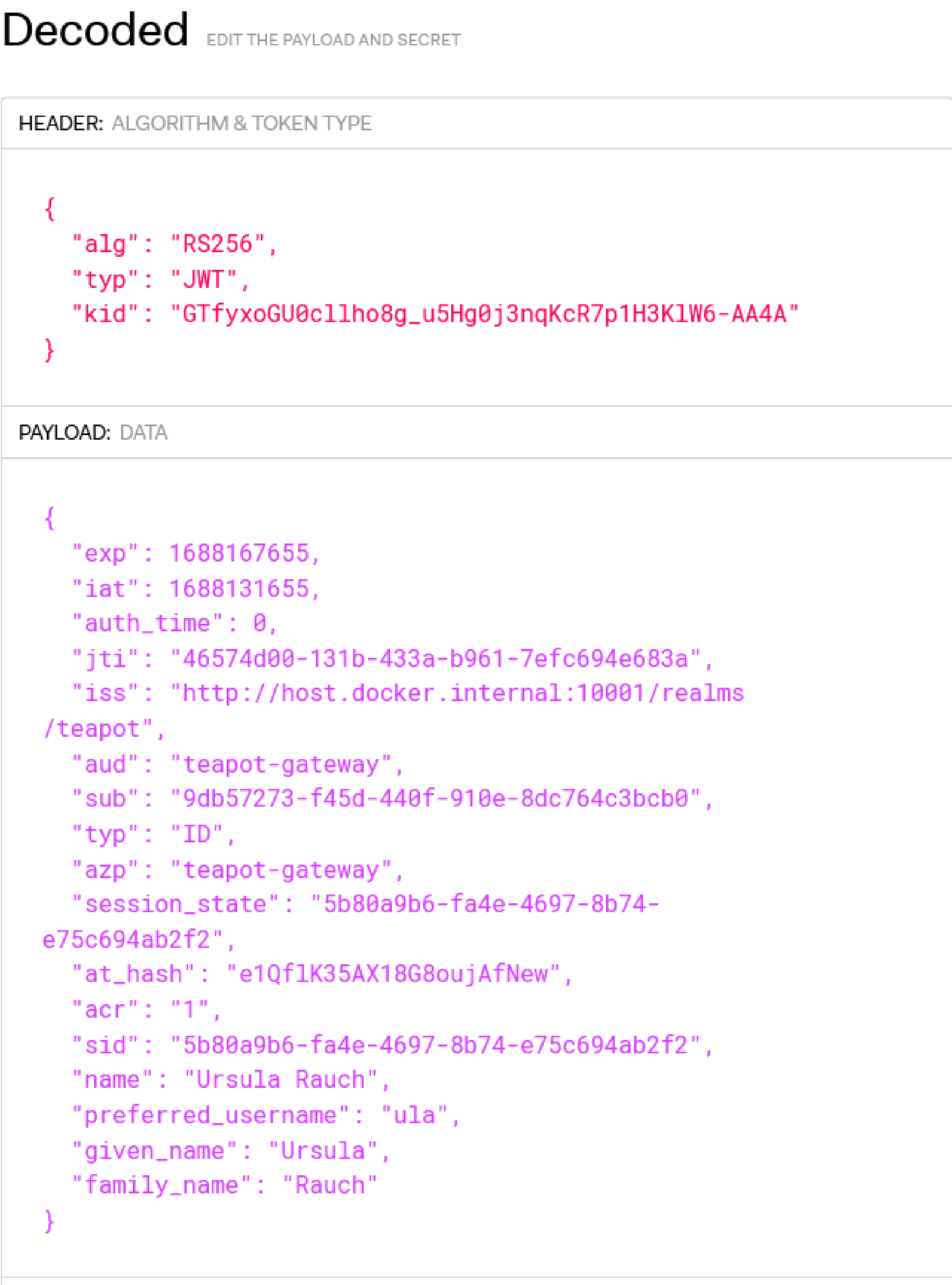

Figure 5.1: Example of an ID token issued by the Keycloak server

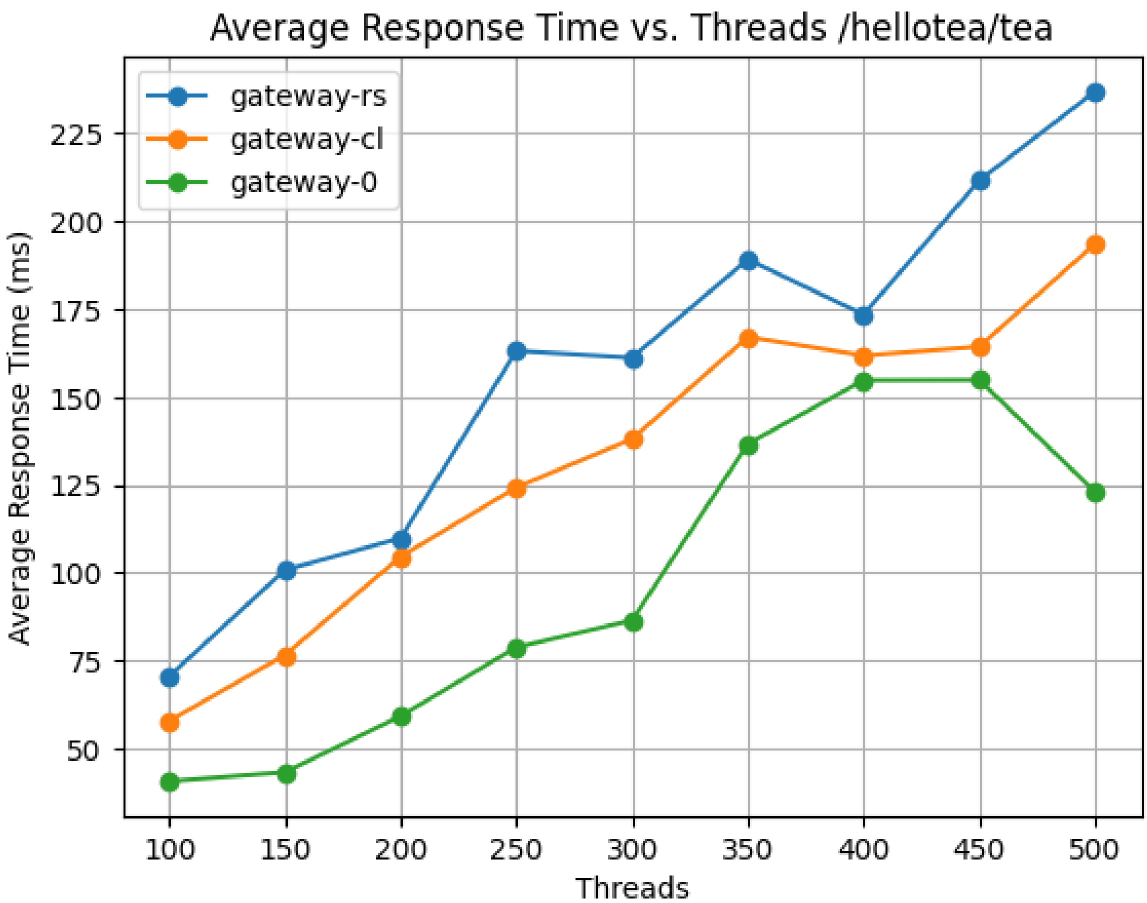

Figure 5.2: Average response times in milliseconds of the `/helloauth` endpoint of all three gateway variants

6 Conclusion and Future Work

In this thesis it has been shown how authentication and authorization can be implemented in a MSA system according to OAuth2 and OIDC specifications using Spring Boot and Keycloak. The outcome was analyzed with Firefox, Postman and Wireshark in oder to determine if the implementation complies with the specification and where further improvement is necessary. The results of this survey made apparent that most basic requirements can be fulfilled with very few configuration steps in the Keycloak admin console, such as creating a realm, a client, and a user, and by overriding the `SecurityFilterChain` in the source code of the resource server and the client (when implemented as part of the MSA). Some requirements were omitted by choice for convenience during the development process, like not protecting communication between services with TLS. On the other hand, the absence of some other required features, like certain token claims and their validation, was revealed only upon further analysis of the generated requests and responses between the services. While it was possible to gather information about how to implement some of these features, this was not the case for the wrong media type (`JWT` instead of `at+jwt`) in the `typ` header value of access tokens issued by Keycloak. As long as the media type can not be distinguished from the media type of ID tokens, a check for the correct `typ` header value, which is required by RFC 9068, can not be included in the resource server without making it unfunctional.

Additionally, logout functionality was implemented, which is not defined in OAuth2 and only mentioned as optional in the OIDC core specification, therefore adherence to the specifications [50] was not analyzed. This could be a possible extension of this research.

Another feature that was implemented but is not part of OAuth2 or OIDC is access control based on user roles. It was not the aim of this thesis to analyze the implementation in detail for compliance with the RBAC standard and it is clear that the presented implementation is a very minimalistic interpretation of RBAC. Keycloak also supports other, more fine-grained methods of access control, like attribute-based access control (ACAB), but the limitation on roles seems sufficient as a PoC within the scope of this thesis. The implementation of ACAB can also be a suggestion for future work.

It is also clear that this thesis, while paying close attention to numerous details in the specifications, was not sufficient to cover all the requirements in detail as it would be necessary for a secure production-ready implementation, not to mention recommendations and best practice. The presented implementation and considerations therefore cover only a selection, which leaves the implementation and analysis of further details for future work. Examples are using a recommended client authentication method like mTLS or JWT, using sender-constrained access tokens, using code challenge/PKCE in the authorization code flow, and of course client communication must be secured with TLS (required by OAuth2).

Another suggestion for future work would be a more thorough and systematic security testing of the implementation. While some tests were performed manually, like using the ID token as access token, manipulating the payload of the access token or sending invalid parameters in the authorization or token request, protection against other attacks was not tested systematically. The OAuth2 specification includes a list of threats, possible attacks, and information about their mitigation.

The second part of the investivation was a comparison of average response times between different implementations of the gateway as a client and as an additional resource server, which requires the frontend application to implement client functionality. Several test rounds of load testing for different numbers of concurrent users showed that the client-gateway was faster than its implementation as resource server. While the more common interpretation seems to be that client in the OAuth2 terminology equals the frontend, this outcome can be interpreted as a challenge to the traditional approach. However, more investigations and considerations are necessary to make a clear statement. Load testing could also be performed for higher numbers of concurrent users and proper security testing should be done as well. It is possible that the client-gateway version requires additional measurements to secure the communication between frontend and gateway.

Bibliography

[1] A. Saboor, M. F. Hassan, R. Akbar, S. N. M. Shah, F. Hassan, S. A. Magsi, and M. A. Siddiqui, "Containerized Microservices Orchestration and Provisioning in Cloud Computing: A Conceptual Framework and Future Perspectives," *Applied Sciences*, vol. 12, no. 12, p. 5793, Jan. 2022. 1

[2] "Microservices use in organizations worldwide 2021," accessed: 2023-07-15. [Online]. Available: https://www.statista.com/statistics/1236823/microservices-usage-per-organization-size/ 1

[3] "Microservices in the enterprise, 2021: Real benefits, worth the challenges. How organizations are finding speed, agility and resiliency through microservices." IBM Market Development & Insights, White Paper, 2021, accessed: 2023-07-15. [Online]. Available: https://www.ibm.com/downloads/cas/OQG4AJAM 1

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," *arXiv*, no. 1606.04036, Apr. 2017, accessed: 2022-12-23. [Online]. Available: http://arxiv.org/abs/1606.04036 1, 6, 8

[5] "Cost of a data breach 2022," Apr. 2023, accessed: 2023-07-14. [Online]. Available: https://www.ibm.com/reports/data-breach 1

[6] "OWASP Top 10 API Security Risks – 2023," accessed: 2023-06-27. [Online]. Available: https://owasp.org/API-Security/editions/2023/en/0x11-t10/ 1, 8

[7] I. T. L. Computer Security Division, "Role Based Access Control | CSRC | CSRC," Nov. 2016, accessed: 2023-07-14. [Online]. Available: https://csrc.nist.gov/projects/role-based-access-control 1

[8] "OAuth 2.0," accessed: 2023-07-18. [Online]. Available: https://oauth.net/2/ 1

[9] P. Billawa, A. B. Tukaram, N. E. D. Ferreyra, J.-P. Steghöfer, R. Scandariato, and G. Simhandl, "SoK: Security of Microservice Applications: A Practitioners' Perspective on Challenges and Best Practices," in *Proceedings of the 17th International Conference on Availability, Reliability and Security*, Aug. 2022, pp. 1–10. 1

[10] D. Hardt, "The OAuth 2.0 Authorization Framework," IETF, RFC 6749, Oct. 2012. 1, 9

[11] M. Jones and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage," IETF, RFC 6750, Oct. 2012. 1, 9

[12] A. Parecki, "It's Time for OAuth 2.1," Dec. 2019, accessed: 2022-12-31. [Online]. Available: https://aaronparecki.com/2019/12/12/21/its-time-for-oauth-2-dot-1 1

[13] D. Hardt, A. Parecki, and T. Lodderstedt, "The OAuth 2.1 Authorization Framework," IETF, Internet Draft draft-ietf-oauth-v2-1-08, Mar. 2023. 1, 9, 10, 11, 12, 13, 33, 34

[14] "Slant - 13 Best microservices frameworks as of 2023," Jul. 2023, accessed: 2023-07-15. [Online]. Available: https://www.slant.co/topics/16161/~microservices-frameworks 1

[15] H. Dhaduk, "Microservices Frameworks for a Scalable Application," Mar. 2022, accessed: 2023-07-15. [Online]. Available: https://www.simform.com/blog/microservices-framework/ 1

[16] "7 Major Reasons to Choose Spring Boot For Microservices Development," Apr. 2023, accessed: 2023-07-15. [Online]. Available: https://www.geeksforgeeks.org/why-to-choose-spring-boot-for-microservices-development/ 1

[17] H. Dinh-Tuan, M. Mora-Martinez, F. Beierle, and S. R. Garzon, "Development Frameworks for Microservice-based Applications: Evaluation and Comparison," no. 2203.07267, Mar. 2022, accessed: 2023-07-15. [Online]. Available: http://arxiv.org/abs/2203.07267 1

[18] "Deprecation of Keycloak adapters," accessed: 2023-06-18. [Online]. Available: https://www.keycloak.org/2022/02/adapter-deprecation 2

[19] Y. Sandeepa, "Using Keycloak with Spring Boot 3.0," Feb. 2023, accessed: 2023-06-17. [Online]. Available: https://medium.com/geekculture/using-keycloak-with-spring-boot-3-0-376fa9f60e0b 2, 27

[20] A. Parecki and D. Waite, "OAuth 2.0 for Browser-Based Apps," IETF, Internet Draft draft-ietf-oauth-browser-based-apps-14, Jun. 2023. 2, 4, 9, 15

[21] Q. Nguyen and O. Baker, "Applying Spring Security Framework and OAuth2 To Protect Microservice Architecture API," *Journal of Software*, vol. 14, no. 6, pp. 257–264, Jun. 2019. 4

[22] A. Chatterjee and A. Prinz, "Applying Spring Security Framework with KeyCloak-Based OAuth2 to Protect Microservice Architecture APIs: A Case Study," *Sensors*, vol. 22, no. 5, p. 1703, Jan. 2022. 4

[23] S. T. Florén, "Implementation and Analysis of Authentication and Authorization Methods in a Microservice Architecture: A Comparison Between Microservice Security Design Patterns for Authentication and Authorization Flows," Master's Thesis, KTH Royal Institute of Technology, Stockholm, SE, 2021. 4

[24] A. Nehme, V. Jesus, K. Mahbub, and A. Abdallah, "Securing Microservices," *IT Professional*, vol. 21, no. 1, pp. 42–49, Jan. 2019. 4

[25] T. Yarygina and A. H. Bagge, "Overcoming Security Challenges in Microservice Architectures," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, Mar. 2018, pp. 11–20. 4

[26] "Backend For Frontend Authentication Pattern with Auth0 and ASP.NET Core," Auth0 - Blog, accessed: 2023-03-15. [Online]. Available: https://auth0.com/blog/backend-for-frontend-pattern-with-auth0-and-dotnet/ 5

[27] "Authentication Patterns and Best Practices For SPAs," accessed: 2023-03-15. [Online]. Available: https://dev.indooroutdoor.io/authentication-patterns-and-best-practices-for-spas 5

[28] J. Lewis and M. Fowler, “Microservices,” Mar. 2014, accessed: 2022-12-23. [Online]. Available: https://martinfowler.com/articles/microservices.html 6

[29] U. Rauch, “Authx-microservices-urauch.pdf,” Bachelor’s Thesis, FH Campus, Wien, AT, Jan. 2023. 6, 10

[30] C. Pahl, “Containerization and the PaaS Cloud,” *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, May 2015. 7

[31] S. Newman, *Monolith to Microservices. Evolutionary Patterns to Transform Your Monolith.* O’Reilly, 2020. 7

[32] A. Akbulut and H. G. Perros, “Performance Analysis of Microservice Design Patterns,” *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, Nov. 2019. 7

[33] I. Bazeniuc and A. Zgureanu, “Information Security in Microservices Architectures,” in *Proceedings of the 11th IC/ECCO.* Technical University of Moldova, oct 2021, accessed: 2022-11-20. [Online]. Available: http://repository.utm.md/handle/5014/20102 7

[34] P. Siriwardena, *Advanced API Security: OAuth 2.0 and Beyond.* Berkeley, CA: Apress, 2020. 7, 8, 9, 10

[35] L. Schwarz, “What Is Authorization? Definition & Examples,” Oracle NetSuite, Apr. 2022, accessed: 2023-08-05. [Online]. Available: https://www.netsuite.com/portal/resource/articles/erp/authorization.shtml 8

[36] E. Shmeleva, “How Microservices are Changing the Security Landscape,” Master’s Thesis, Aalto University, Espoo, FI, nov 2020. 8

[37] Z. Lu, D. T. Delaney, and D. Lillis, “A Survey on Microservices Trust Models for Open Systems,” *IEEE Access*, vol. 11, pp. 28 840–28 855, 2023. 8

[38] A. Nehme, V. Jesus, K. Mahbub, and A. Abdallah, “Fine-Grained Access Control for Microservices,” in *Foundations and Practice of Security*, ser. Lecture Notes in Computer Science, N. Zincir-Heywood, G. Bonfante, M. Debbabi, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2019, pp. 285–300. 8

[39] D. Ferraiolo and R. Kuhn, “Role-Based Access Controls,” in *Proceedings of the 15th National Computer Security Conference.* National Institute of Standards and Technoloty, Oct. 1992, pp. 554–563. 9

[40] B. Kelechava, “Role Based Access Control (RBAC),” May 2018, accessed: 2023-07-16. [Online]. Available: https://blog.ansi.org/2018/05/role-based-access-control-rbac-incits-359/ 9

[41] N. Barbettini, “OAuth 2.0 and OpenID Connect (in plain English),” Feb. 2018, accessed: 2022-12-18. [Online]. Available: https://www.youtube.com/watch?v=996OiexHze0 9

[42] W. Denniss, J. Bradley, M. Jones, and H. Tschofenig, “OAuth 2.0 Device Authorization Grant,” IETF, RFC 8628, Aug. 2019. 9

[43] M. Jones, A. Nadalin, B. Campbell, J. Bradley, and C. Mortimore, “OAuth 2.0 Token Exchange,” IETF, RFC 8693, Jan. 2020. 9

[44] B. Campbell, J. Bradley, and H. Tschofenig, “Resource Indicators for OAuth 2.0,” IETF, RFC 8707, Feb. 2020. 9

[45] N. Sakimura, J. Bradley, and M. Jones, "The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)," IETF, RFC 9101, Aug. 2021. 9

[46] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations," IETF, RFC 6819, Jan. 2013. 9

[47] N. Sakimura, J. Bradley, and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients," IETF, RFC 7636, Sep. 2015. 9

[48] T. Lodderstedt, J. Bradley, A. Labunets, and D. Fett, "OAuth 2.0 Security Best Current Practice," IETF, Internet Draft draft-ietf-oauth-security-topics-23, Jun. 2023. 9, 10, 15, 27

[49] W. Denniss and J. Bradley, "OAuth 2.0 for Native Apps," IETF, RFC 8252, Oct. 2017. 9

[50] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "OpenID Connect Core 1.0," OpenID Foundation (OIDF), Aug. 2014. 9, 13, 33, 37

[51] S. Yang and J. Hu, "Research on Unified Authentication and Authorization in Microservice Architecture," in *2020 IEEE 20th International Conference on Communication Technology (ICCT)*, Oct. 2020, pp. 1169–1173. 12

[52] N. Jackson, "Microservice Authentication and Authorization," Feb. 2019, accessed: 2022-12-18. [Online]. Available: https://www.youtube.com/watch?v=ZjPF8yZ83Wo 12

[53] V. Bertocci, "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens," IETF, RFC 9068, Oct. 2021. 12, 13, 33, 34

[54] M. Jones, N. Sakimura, and J. Bradley, "OAuth 2.0 Authorization Server Metadata," IETF, RFC 8414, Jun. 2018. 13

[55] P. Siriwardena and N. Dias, *Microservices Security in Action.* Shelter Island, NY: Manning, 2020. 15

[56] A. Barabanov and D. Makrushin, "Authentication and authorization in microservice-based systems: Survey of architecture patterns," *arXiv*, no. 2009.02114, Sep. 2020, accessed: 2023-03-15. [Online]. Available: http://arxiv.org/abs/2009.02114 15

[57] V. Bertocci and B. Campbell, "Token Mediating and session Information Backend For Frontend," IETF, Internet Draft draft-bertocci-oauth2-tmi-bff-01, Apr. 2021. 15

[58] "Sam Newman - Backends For Frontends," accessed: 2023-04-20. [Online]. Available: https://samnewman.io/patterns/architectural/bff/ 15

[59] "Introduction to Spring Framework," accessed: 2023-06-06. [Online]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html 18

[60] C. Walls, *Spring Boot in Action.* Shelter Island, NY: Manning, 2016. 18, 19

[61] "Spring Core Features," accessed: 2023-06-06. [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/features.html 19

[62] "OAuth 2.0 Resource Server JWT," Spring Security, accessed: 2023-06-16. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html 19, 24, 26, 28, 34

[63] "Spring Cloud Gateway," accessed: 2023-06-05. [Online]. Available: https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/ 20, 22

[64] "Getting Started | Spring Boot and OAuth2," accessed: 2023-06-06. [Online]. Available: https://spring.io/guides/tutorials/spring-boot-oauth2/ 21

[65] N. Sakimura, J. Bradley, and E. Jay, "OpenID Connect Discovery 1.0," OpenID Foundation (OIDF), Nov. 2014. 21

[66] G. Roza, "WebClient and OAuth2 Support | Baeldung," Jan. 2019, accessed: 2023-06-05. [Online]. Available: https://www.baeldung.com/spring-webclient-oauth2 22

[67] "Advanced Configuration," Spring Security, accessed: 2023-06-19. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/oauth2/login/advanced.html 23

[68] "How To Fix Keycloak Oauth2 OIDC Logout With Spring Cloud Gateway," Aug. 2022, accessed: 2023-06-12. [Online]. Available: https://refactorfirst.com/fix-keycloak-oauth2-oidc-logout-with-spring-cloud-gateway 23

[69] G. Roza, "Spring Security and OpenID Connect | Baeldung," Mar. 2017, accessed: 2023-06-13. [Online]. Available: https://www.baeldung.com/spring-security-openid-connect 23

[70] M. Jones, B. de Medeiros, N. Agarwal, N. Sakimura, and J. Bradley, "OpenID Connect RP-Initiated Logout 1.0," OpenID Foundation (OIDF), Sep. 2022. 23

[71] "Authentication Persistence and Session Management," Spring Security, accessed: 2023-07-16. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/authentication/session-management.html 24

[72] S. Wagde, "OAuth 2.0 Resource Server With Spring Security 5 | Baeldung," Aug. 2020, accessed: 2023-06-16. [Online]. Available: https://www.baeldung.com/spring-security-oauth-resource-server 24, 26

[73] "OAuth 2.0 Resource Server," Spring Security, accessed: 2023-06-16. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/index.html 24, 26

[74] "BearerTokenAuthenticationEntryPoint.java," Spring Security, Jun. 2023, accessed: 2023-06-16. [Online]. Available: https://github.com/spring-projects/spring-security/blob/61e2bae7f2a3127c6b0ca662a2611ac8fff091f2/oauth2/oauth2-resource-server/src/main/java/org/springframework/security/oauth2/server/resource/web/BearerTokenAuthenticationEntryPoint.java 25

[75] "BearerTokenAuthenticationToken," (Spring-Security-Docs 6.1.1 API), accessed: 2023-06-25. [Online]. Available: https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/oauth2/server/resource/authentication/BearerTokenAuthenticationToken.html 25

[76] “Principal,” accessed: 2023-06-25. [Online]. Available: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/security/Principal.html 25

[77] “Authentication,” (Spring-Security-Docs 6.1.1 API), accessed: 2023-06-25. [Online]. Available: https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/Authentication.html 25

[78] “Servlet Authentication Architecture,” Tech. Rep., accessed: 2023-06-17. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/authentication/architecture.html 25

[79] “OAuth2ResourceServerConfigurer,” (Spring-Security-Docs 6.1.0 API), accessed: 2023-06-18. [Online]. Available: https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/configurers/oauth2/server/resource/OAuth2ResourceServerConfigurer.html 27

[80] “Cross Site Request Forgery (CSRF),” Spring Security, accessed: 2023-07-18. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/exploits/csrf.html 27

[81] Ch4mpy, “Spring-addons/samples/tutorials/bff,” GitHub Repository, accessed: 2023-06-26. [Online]. Available: https://github.com/ch4mpy/spring-addons 27

[82] “Server Administration Guide,” Keycloak, accessed: 2023-06-24. [Online]. Available: https://www.keycloak.org/docs/latest/server_admin/ 28

[83] “Authorization Architecture,” Spring Security, accessed: 2023-07-16. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/authorization/architecture.html 28

[84] ch4mp, “Answer to "Use Keycloak Spring Adapter with Spring Boot 3",” Stack Overflow, Nov. 2022, accessed: 2023-06-23. [Online]. Available: https://stackoverflow.com/a/74572732 29, 30

[85] “Apache JMeter - User’s Manual: Elements of a Test Plan,” accessed: 2023-07-16. [Online]. Available: https://jmeter.apache.org/usermanual/test_plan.html 32

[86] “Authorization Services Guide,” Keycloak, accessed: 2023-06-30. [Online]. Available: https://www.keycloak.org/docs/latest/authorization_services/ 33

[87] “Client Authentication Support,” Spring Security, accessed: 2023-06-30. [Online]. Available: https://docs.spring.io/spring-security/reference/servlet/oauth2/client/client-authentication.html 33

List of Figures

Listings

Appendix

Network traffic produced by a request to a protected resource (shortened Wireshark capture)

The protocol is HTTP 1.1 and host is localhost where not marked differently.

1. Source Port: 50965, Dest. Port: 8181	
Request	Response
`GET http://localhost:8181/listteas`	`302 Found Location: /oauth2/authorization/ keycloak-gateway-client`
Unauthenticated user wants to view a list of teas and sends request to dedicated endpoint at the gateway (client)	Gateway refers the user to a different endpoint at the gateway

2. Source Port: 50965, Dest. Port: 8181	
Request	Response
`GET http://localhost:8181/listteas`	`302 Found Location: /oauth2/authorization/ keycloak-gateway-client`
Unauthenticated user wants to view a list of teas and sends request to dedicated endpoint at the gateway (client)	Gateway refers the user to a different endpoint at the gateway

3. Source Port: 50965, Dest. Port: 8181	
Request	Response

`GET /oauth2/authorization/`
`keycloak-gateway-client`
[truncated]
`response_type=code`
`client_id=teapot-gateway`
`scope=openid`
`state=FczD4rvwKxMMyYRC0joaa-oAr`
`-GMbaG2rbuW76gE72k%3D`
`redirect_uri=http://localhost...`
`nonce=...` omitted
	`302 Found`
	`[truncated]Location:`
	`http://host.docker.internal:`
	`10001/realms/teapot/protocol/`
	`openid-connect/auth...`

User agent sends request to the new endpoint	Gateway refers user agent to Keycloak's authorization endpoint with the authorization request

—Additional requests to load login page omitted—

4. Source Port: 50966, Dest. Port: 10001	
Request	Response

`GET /oauth2/authorization/keycloak -gateway-client`
 	
`response_type=code`
`client_id=teapot-gateway`
`scope=openid`
`state=FczD4rvwKxMMyYRC0joaa-oAr-GMbaG2rbuW76gE72k%3D`
`redirect_uri=http://localhost: /login/oauth2/code/keycloak-gateway-client`
`nonce=BVUZ0gzd1P-FjEqiFWv8gLspn EEdaOpvvqm3hgiAriI`
	`200 OK (text/html)`

User agent sends the client's authorization request to Keycloak's authorization endpoint	Keycloak responds with the login page to authenticate the user

5. Source Port: 50971, Dest. Port: 10001	
Request	Response

`[truncated]POST /realms/teapot/login-actions/ authenticate...`
(request query parameters omitted)
`Form item:  "username" = "ula"`
`Form item:  "password" = "pass"`
`Form item:  "credentialId" = ""`
	`302 Found`
	`[truncated]Location: http://localhost:8181/login/ oauth2/code/keycloak-gateway -client?state=FczD4rvwKxMMyY RC0joaa-oAr-GMbaG2rbuW76gE...`

User sends login data (username and password in payload)	Keycloak sends the user agent back to the gateway with the authorization code and state parameters (omitted - see below for full list of parameters)

6. Source Port: 50965, Dest. Port: 8181
Request (Response in 10.)

`GET [truncated]/login/oauth2/code/ keycloak-gateway-client...`
 	
`state=FczD4rvwKxMMyYRC0joaa-oAr-GMbaG2rbuW76gE72k%3D`
`session_state=4165bd59-749c-4ed3-a210-ecd4a6f928f5`
`code=c49d4ea6-01c0-4f30-8388-48f338bdbd20.4165bd59-749c-4ed3-a210-ecd4a6f928f5.672c0413-eba3-4481-abb5-27f75a71727e`

User agent sends the code back to the gateway

<table>
<tr><td colspan="2">7. Source Port: 50976, Dest. Port 10001</td></tr>
<tr><td>Request</td><td>Response</td></tr>
<tr><td><code>POST
/realms/teapot/protocol/openid
-connect/token
(application/x-www-form
-urlencoded)

Authorization: Basic
dGVhcG90LWdhdGV3YXk6U2ZBN3RCZ
FM5V2RrVnozRFlHcG5NOTY3UkFZTUtFU2s

Form item: "grant_type" =
"authorization_code"
Form item: "code" =
"c49d4ea6-01c0
-4f30-8388-48f338bdbd20.4165bd59
-749c-4ed3-a210-ecd4a6f928f5.672
c0413-eba3-4481-abb5-27f75a71727e"
Form item: "redirect_uri" =
"http://localhost:8181/login/oauth2/
code/keycloak-gateway-client"</code></td><td><code>200 OK , JavaScript Object
Notation (application/json)

"access_token":</code> (value omitted)
<code>"expires_in": "36000"
"refresh_expires_in": "1800"
"refresh_token":</code> (value omitted)
<code>"token_type": "Bearer"
"id_token":</code> (value omitted)
<code>"not-before-policy": "0"
"session_state":
"4165bd59-749c-4ed3-a210
-ecd4a6f928f5"
"scope": openid profile email</code></td></tr>
<tr><td>Gateway sends token request to Keykloaks token endpoint, with grant type, code and redirect-uri in the payload</td><td>Keycloak sends token response (in payload) to the gateway</td></tr>
</table>

<table>
<tr><td colspan="2">8. Source Port: 50977, Dest. Port: 10001</td></tr>
<tr><td>Request</td><td>Response</td></tr>
<tr><td><code>GET /realms/teapot/protocol/
openid-connect/certs</code></td><td><code>200 OK , JavaScript Object
Notation (application/json)</code></td></tr>
<tr><td>Gateway requests the JWKS from Keycloak</td><td>Keycloak returns JWKS in payload (omitted)</td></tr>
</table>

<table>
<tr><td colspan="2">9. Source Port: 50976, Dest. Port: 10001</td></tr>
<tr><td>Request</td><td>Response</td></tr>
<tr><td><code>GET /realms/teapot/protocol/
openid-connect/userinfo

[truncated] Authorization:
Bearer eyJhbGciOiJSUzI1NiIsInR
5cCIgOiAiSldUIiwia2lkIiA6ICJHV...</code></td><td><code>200 OK , JavaScript Object
Notation (application/json)</code></td></tr>
<tr><td>Gateway requests additional user information from Keykloak’s userinfo endpoint with the freshly issued access token</td><td>Keycloak returns user info (omitted)</td></tr>
</table>

<table>
<tr><td>10. Source Port: 8181, Dest. Port: 50965</td></tr>
<tr><td>Response to 6.</td></tr>
<tr><td><code>302 Found Location: /listteas
Set-Cookie: SESSION=25b0ce5f-b4b2-49c8-8b39-5a671f3beff2</code></td></tr>
<tr><td>Gateway refers user to its own <code>/listteas</code> endpoint a second time and tells the browser to set a session cookie</td></tr>
</table>

11. Source Port: 50965, Dest. Port: 8181
Request (Response in 13.)
`GET /listteas` `Cookie:  SESSION=25b0ce5f-b4b2-49c8-8b39-5a671f3beff2`
Browser sends new request to gateway's `/listteas` endpoint together with the new session cookie

12. Source Port: 50978, Dest. Port: 8184	
Request	Response

`GET /teas/getall`
`[truncated]Authorization:`
`Bearer`
`eyJhbGciOiJSUzI1NiIsInR5cCIg`
`OiAiSldUIiwia2lkIiA6ICJHVGZ5e...`
	`200 OK , JavaScript Object`
	`Notation (application/json)`

Gateway forwards request for list of teas to Tea service with the access token in the `Authorization` header	Tea service responds to gateway with list of teas in payload (omitted)

13. Source Port: 8181, Dest. Port: 50965
Response to 11.
`200 OK , JavaScript Object Notation (application/json)`
Gateway forwards the response from the Tea service to the user agent.