Karlsruhe Institute of Technology

SCC
Steinbuch Centre for Computing

mytoken - OpenID Connect Tokens for Long-term Authorization

Master's Thesis of

Gabriel Zachmann

at the Department of Informatics
Steinbuch Centre for Computing (SCC)

Reviewer:	Prof. Dr. Achim Streit
Second reviewer:	Prof. Dr. Bernhard Neumair
Advisor:	Dr. Marcus Hardt

02. December 2020 – 05. May 2021

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Remchingen, 28. April 2021

. . . . . .

(Gabriel Zachmann)

Abstract

OpenID Connect is an important key component of many modern Authentication and Authorization Infrastructures. It is mostly used within web browsers, but command line and API usages see rising adoption. From this emerges the need for good command line tooling. In previous work we developed a local tool for managing OpenID Connect access tokens on the command line. However, this tool is less suitable if access tokens need to be used over extended periods of time from multiple hosts. This need arises especially with compute jobs which typically do not run on the user's computer. In scenarios in which compute jobs need to authenticate via OpenID Connect access tokens to load and store data or to access other resources, the short lifetime of OpenID Connect access tokens may pose a problem. It is very common that compute jobs run longer than the lifetime of a single access token. Current mechanisms provided by OpenID Connect and existing software tools do not address this problem sufficiently.

In this thesis we designed and implemented a token service called mytoken that addresses this problem by providing a mechanism to provide fresh OpenID Connect access tokens over an extended period of time. The mytoken service introduces a new token type called *mytoken*. These mytokens can be used for easily obtaining access tokens on any internet-connected device via the mytoken server. Mytokens can be transferred easily to other devices. Mytokens may be used instead of OpenID Connect access tokens to start compute jobs. They then can be used by job submission systems as well as from inside running compute jobs to obtain as many access tokens as specified in the mytoken. For a fine grained balance between security and usefulness, we have implemented *capabilities* and *restrictions*. These can be used to restrict each mytoken so that it has exactly the power it needs, and to limit the impact if stolen. Restrictions allow limiting the usage of a mytoken to a set of time spans, to certain locations (countries as well as ip addresses or subnets), in the number of usages, and the scopes and audiences of access tokens that can be obtained. By using multiple restriction *clauses*, a single mytoken can be used even in complex use cases that demand different privileges over time. A typical example for this is job submission, where initially privileges for starting a job and reading input data are required. Throughout the run-time of a job, usually no privileges are required. At the end of the job output data needs to be written. Using the introduced restrictions, a single mytoken can be used for all of these steps, while in each step only the required privileges are active, which reduces the attack surface greatly, and thereby poses a promising solution for long running jobs.

Zusammenfassung

OpenID Connect ist eine wichtige Schlüsselkomponente in vielen modernen Authentifizierungs- und Autorisierungsinfrastrukturen. Es wird hauptsächlich in Webbrowsern verwendet, aber auch die Verwendung auf der Kommandozeile und in APIs findet immer mehr Verbreitung. Daraus ergibt sich der Bedarf für gute Kommandozeilen-Werkzeuge. In einer früheren Arbeit haben wir ein Werkzeug für die Verwaltung von OpenID Connect Access Tokens auf der Kommandozeile entwickelt. Dieses Werkzeug ist jedoch weniger geeignet, wenn Access Tokens über längere Zeiträume auf mehreren Rechnern verwendet werden müssen. Diese Notwendigkeit ergibt sich insbesondere bei Rechenjobs, die typischerweise nicht auf dem Computer des Benutzers laufen. In Szenarien, in denen Rechenjobs sich über OpenID Connect Access Tokens authentifizieren müssen, um Daten zu laden und zu speichern oder um auf andere Ressourcen zuzugreifen, führt die kurze Gültigkeitsdauer von OpenID Connect Access Tokens zu Problemen. Es ist üblich, dass Rechenjobs wesentlich länger dauern als ein einzelner Access Token gültig ist. Die aktuellen Mechanismen, die OpenID Connect bietet, und bestehende Software-Werkzeuge lösen dieses Problem nicht zufriedenstellend.

In dieser Arbeit haben wir einen Token-Dienst - genannt Mytoken - entworfen und implementiert, der dieses Problem löst, indem ein Mechanismus bereitgestellt wird, der es ermöglicht gültige OpenID Connect Access Tokens über einen längeren Zeitraum abzurufen. Mit so genannten *Mytokens* können Access Tokens auf jedem mit dem Internet verbundenen Gerät einfach über den Mytoken-Server abgerufen werden. Mytokens könnten anstelle von OpenID Connect Access Tokens verwendet werden, um Rechenjobs zu starten. Sie können dann sowohl von Job-Submission-Systemen als auch von laufenden Rechenjobs verwendet werden, um so viele Access Tokens abzurufen, wie es durch den Mytoken erlaubt ist. Um ein gutes Gleichgewicht zwischen Sicherheit und Benutzbarkeit zu erreichen, haben wir so genannte *Capabilities* und *Restrictions* implementiert. Diese können verwendet werden, um Mytokens so einzuschränken, dass jeder genau die benötigten Rechte hat, und um gleichzeitig die Auswirkungen eines Diebstahls zu begrenzen. Mit Restrictions kann die Verwendung eines Mytokens auf eine oder mehrere Zeitspannen, auf bestimmte Standorte (Länder sowie IP-Adressen oder Subnetze), und in der Anzahl der Nutzungen beschränkt werden. Außerdem kann festgelegt werden, dass Access Tokens, die mit dem Mytoken abgerufen werden, nur bestimmte Scopes und Audiences haben können. Durch die Verwendung mehrerer so genannter *Restriction Clauses* kann ein einziger Mytoken auch in komplexen Anwendungsfällen verwendet werden, in denen zu unterschiedlichen Zeiten unterschiedliche Berechtigungen benötigt werden. Ein typisches Beispiel hierfür ist das Starten von Rechenjobs. Hierbei werden zu Beginn Berechtigungen zum Starten eines Jobs und zum Lesen von Eingabedaten benötigt; am Ende müssen die Ausgabedaten geschrieben werden. Mit den Restrictions kann hierfür ein einziger Mytoken verwendet werden, wobei in jedem Schritt nur die benötigten Privilegien nutzbar sind, wodurch die Angriffsfläche stark reduziert wird. Somit stellen Mytokens eine vielversprechende Lösung für lang laufende Jobs dar.

Acknowledgment

First and foremost, I would like to thank God for giving me the opportunity, knowledge, and ability to complete my thesis. I'm thankful for all his blessings.

Furthermore, I want to thank my supervisor Marcus Hardt for his feedback, guidance, and advice throughout this thesis and my whole study. I would also like to thank him for the trust and appreciation he has placed in me and the freedom and flexibility this has given me. In addition, I also want to thank my other colleagues for their advice throughout my work time at KIT.

I also want to thank my wife for always being at my side and motivating me to sit down and write on the thesis. I appreciate her and my family's support throughout this thesis and my whole study.

Contents

List of Figures

List of Listings

Acronyms

AAI Authentication and Authorization Infrastructure. i, 1, 2, 5, 7, 8, 22

AARC Authentication and Authorisation for Research Collaborations. 6, 7

API Application Programming Interface. i, iii, 44

AS Authorization Server. 9

CA Certificate Authority. 7

DFN Deutsches Forschungsnetz. 8

EGI European Grid Infrastructure. 8, 17, 21, 22, 24, 47, 48

EOSC European Open Science Cloud. 1

HBP Human Brain Project. 1

HIFIS Helmholtz Federated IT Services. 1, 8

HMAC Hash-based MAC. 41

HPC High Performance Computing. 8, 27

https HyperText Transfer Protocol Secure. 8, 13, 28

IdP Identity Provider. 1, 7, 8

JSON JavaScript Object Notation. 8, 18, 27, 30, 45, 57, 58

JWT JSON Web Token. 11, 12, 26–29, 33, 35, 39–41, 57

KIT Karlsruhe Institute of Technology. 5, 22, 49

MAC Message Authentication Code. 41

NHS National Health Service. 8

OIDC OpenID Connect. i, 1, 2, 5, 6, 8–15, 17–24, 27–29, 32–35, 37, 38, 41, 44, 47, 50, 52–56

OP OpenID Provider. 1, 9–14, 16–23, 25, 27, 32, 33, 38, 39, 45, 47, 48, 50, 52–56

PKI Public Key Infrastructure. 7

PR Protected Resource. 9, 10, 17, 19

RP Relying Party. 9, 17

SAML Security Assertion Markup Language. 6–8

SCC Steinbuch Centre for Computing. 5, 47

SP Service Provider. 7, 8

ssh Secure Shell. 23, 27, 53

SSO Single Sign-On. 7

TLS Transport Layer Security. 7

URL Uniform Resource Locator. 13, 14, 26, 27, 32, 33, 42, 53

UUID Universally Unique Identifier. 27

WLCG Worldwide LHC Computing Grid. 1, 5, 7, 8, 48, 50, 51, 53, 54

XML Extensible Markup Language. 7

1. Introduction

OpenID Connect (OIDC) has become a widely used protocol in modern Authentication and Authorization Infrastructures (AAIs). Many social Identity Providers (IdPs) and enterprises like Google, Apple, Microsoft, Telecom, or IBM build on OIDC [24, 43]. Also in the national and international research environment many projects, such as Helmholtz Federated IT Services (HIFIS), the Human Brain Project (HBP), or the European Open Science Cloud (EOSC), use OIDC [45]. Even the Worldwide LHC Computing Grid (WLCG) which was built on X.509 certificates is moving to token-based authorization [44]. This wide adaption of OIDC shows the importance of the protocol. But this wide adaption also demands good tooling.

While the usage of OIDC on the web is user-friendly, handling access tokens within non-web use cases still does not come up to its full potential. In a previous work we developed a local service named oidc-agent [47, 48] that enables users to manage OIDC tokens on the command line. While this was an important step, it did not solve what we call the *long-running-jobs problem*, on which we elaborate in the following.

1.1. Motivation

OIDC access tokens are *Bearer tokens* [42, 26, 29] which means that they can be passed around and any entity in possession of the token can use it to get access to a protected resource. Among other reasons this is why access tokens are short-lived. Their lifetime is usually in the range of minutes to a few hours, e.g. the default for many OpenID Providers (OPs) in the academic world is one hour. While a short lifetime is strongly desirable from a security perspective, it also reduces the usability. It is unquestionable that there are use cases where an entity needs to be able to access a resource over a longer period than the lifetime of an access token. OIDC's mechanism to refresh access tokens with a *Refresh Token* after the access token expired solves this problem in many cases, but not in all of them. Also, other mechanisms that could mitigate the problem are insufficient or not implemented. We will discuss them in subsection 2.2.5.

A good example for a use case where access tokens are required to function over a longer period of time and current mechanism are insufficient are long-running compute jobs: We assume that researchers can start compute jobs on a compute cluster with an OIDC access token and that other resources can also be accessed through OIDC. A typical compute job can be divided into three steps:

1. Load data.
2. Do computation on the data.

3. Store the results.

We assume that data is loaded in the beginning of the compute job. In order to load the data from the storage server an access token is required. The same is true for writing back the results after the computation is done. While it is quite possible that the job requires loading additional data and write back partial results in between, the problem becomes clear once output data is written back: Such computations will often run for several hours and even days, and will almost always take longer than the lifetime of an access token. While loading the data at the beginning with the access token used to start the compute job might be possible, writing back the result at the end will most-likely not work, because the access token expired in-between. It is also possible, that the job has been queued and the access token is already expired when the job starts. The long-running-jobs problem requires either a token that can be easily refreshed or a token that is valid long enough. However, current mechanisms provided by OIDC and current provider implementations satisfy neither of these options as argued in subsection 2.2.5.

1.2. Goal

The goal of this thesis was to develop a solution for the long-running-jobs problem. A token service should be developed, through which users can easily obtain access tokens on any of their machines, including long running compute jobs. This implies that an access token can be obtained with a single token-like string alone.

The service should overcome the shortages of OIDC access tokens, in particular the constraint that the lifetime of an access token cannot be freely chosen by a user. With this service the user should have full control over the issued tokens.

After developing a design the service should be implemented, as well as a basic command line client that can be used to easily obtain access tokens.

1.3. Structure of the Thesis

This thesis is subdivided into several chapters: State of the art in chapter 2, design aspects in chapter 3, implementation aspects in chapter 4, an evaluation in chapter 5, and finally the conclusion in chapter 6. In particular, the individual chapters present the following:

chapter 2 introduces foundations for this work and includes an overview over AAIs and commonly used protocols. We will also present OIDC more detailed and argue why mechanism provided by OIDC are not suitable to solve the long-running-jobs-problem. We then introduce related software tools that could be used.

chapter 3 defines the requirements for the service that should be developed and introduces the important concepts of the mytoken service. These are mytokens, capabilities, and restrictions. We also describe the different endpoints of the service.

chapter 4 focuses on different implementation aspects. We cover the encryption of refresh tokens, support for high availability, and introduce the command line client.

chapter 5 evaluates the developed mytoken service against the previously defined requirements and related software.

chapter 6 closes the thesis with a summary and an outlook on possible future work.

2. State of the Art

In this chapter we discuss the foundations of current Authentication and Authorization Infrastructures (AAIs). We present a general overview of modern federated AAIs and shortly describe some protocols that are commonly used. We then focus on OpenID Connect, the lifetime-problem of OpenID Connect (OIDC) access tokens in the context of long running compute jobs, and how it could be solved with mechanisms provided by the OIDC protocol. In the end we introduce related software that provides OIDC access tokens to users on the command line and that also could be used to solve the long-running-jobs problem.

2.1. Foundations

This work aims at extending the lifetime of OIDC access tokens. In this section we describe the foundations for this work: Authentication and Authorization Infrastructures and commonly used technologies.

2.1.1. Authentication and Authorization Infrastructures

Research nowadays is commonly not done by a single researcher, but multiple researchers working together. Often, research involves multiple institutions from different countries and also interdisciplinary research is common [37]. The EU research and innovation program *Horizon 2020* [11] provided nearly €80 billion funding over a period of seven years (2014-2020). The follow-up program *Horizon Europe* [10] will increase the budget to around €95 billion for 2021-2027. Also researchers of the Karlsruhe Institute of Technology (KIT) and Steinbuch Centre for Computing (SCC) are involved in multiple EU-funded projects, like *EGI-ACE* [23], *EOSC-hub* [20], or *EOSC-synergy* [21]. In these projects researchers not only collaborate, but their goal is to empower researchers to collaborate. Another great example of research collaboration is *CERN* [8] and the Worldwide LHC Computing Grid (WLCG) [9].

All of this shows how important collaboration in research is. However, collaboration also requires the right infrastructures. One part being access control through Authentication and Authorization Infrastructures. The AAI-basis for collaboration between universities in the state of Baden-Württemberg is bwIDM [31]. It enables researches and students of the different universities to access IT services, such as bwUniCluster or bwSync&Share, on universities throughout the state in a consistent and secure way.

While Authentication and Authorization are often perceived as one step from a user's perspective, these are separate actions:

Definition 1 (Authentication). Authentication is the act of validating that a user is whom they claim to be.

Definition 2 (Authorization). Authorization is the act of giving a user permission to access resources or perform actions.

The Authentication and Authorisation for Research Collaborations (AARC) project [40] proposed a blueprint architecture [34] for research collaborations. This blueprint architecture is visualized in Figure 2.1. It defines fives layers of components:

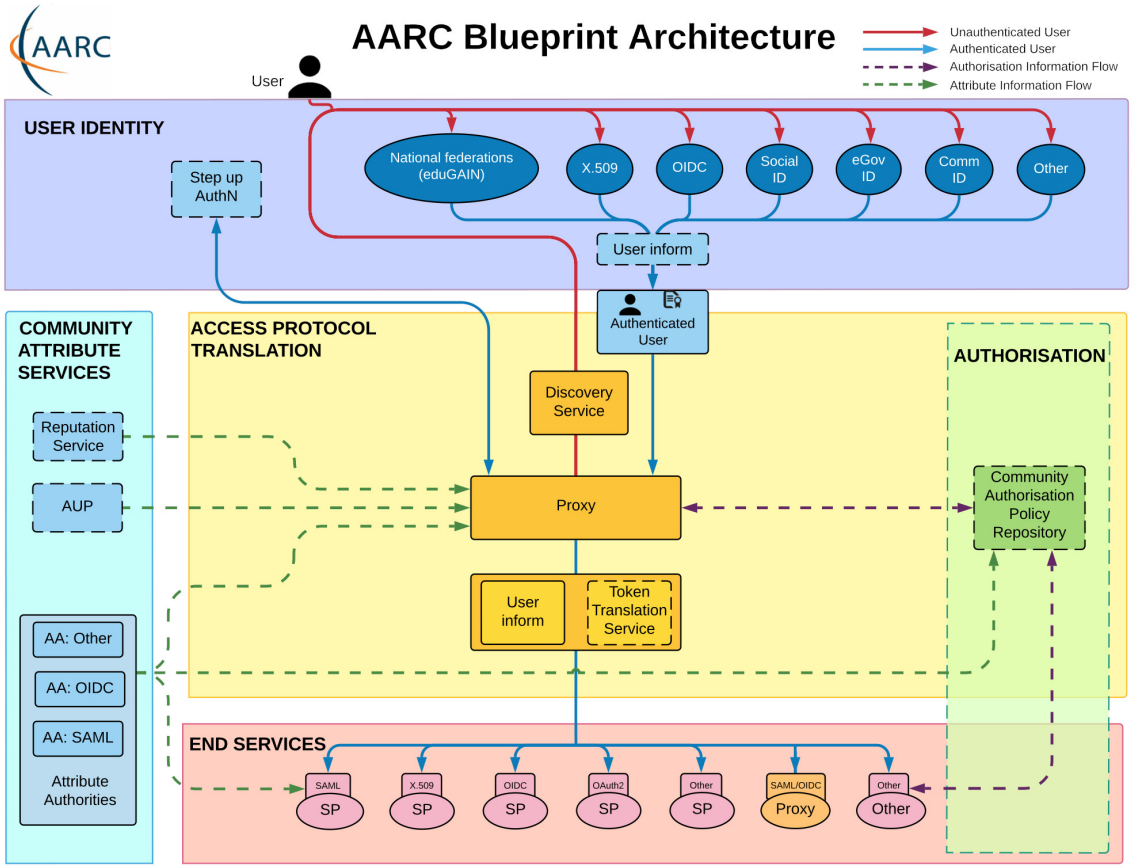


Figure 2.1.: Component layers of the AARC Blueprint Architecture [34]

User Identity The user identity layer contains services that handle identification and authentication of users. Commonly, authentication services are Security Assertion Markup Language (SAML)-, OIDC-, or OAuth2-providers, but also certification authorities.

Community Attribute Services The community attribute services layer holds services that provide user information (attributes). This information is added on top of any information that might be provided by the identity provider from the user identity layer.

Access Protocol Translation Different end services might require different authentication technologies. The access protocol translation layer addresses this need by introducing a token translation service that can translate between different technologies. This layer also contains an SP-IdP-Proxy as a single integration point between identity providers from the user identity layer and service providers from the end services layer. For the identity providers the proxy acts as a service provider, while for service providers it acts as an identity provider.

Authorization The authorization layer contains components that handle the authorization decision, i.e. they control the access to service providers.

End Services The end services layer contains Service Providers (SPs) a user wants to use. The access to these services is usually protected and different services might use different technologies.

The AARC blueprint architecture is a general AAI architecture that does not enforce usage of certain protocols. It can be used with different protocols and it should have become clear that the infrastructure can contain service providers that use different protocols for authorization. Therefore, we will shortly introduce important authorization mechanisms in the following sections.

2.1.2. X.509 Certificates

X.509 [12] certificates are a way to authenticate using public key cryptography. In a Public Key Infrastructure (PKI) each entity has a key pair consisting of a public and private key. For authentication, an entity can create a signature with its private key. Anyone with access to the public key can verify the signature. It is obvious that the private key must be kept secret while the public key is publicly available. In a PKI Certificate Authorities (CAs) are trusted entities that issue certificates to entities and provide assurance about the entity identified in a certificate.

X.509 certificates are used with TLS in the internet to ensure a secure connection to a web server. But it is also possible to use client certificates to authenticate clients. This has been used by WLCG for 20 years [7, 2].

2.1.3. SAML

The Security Assertion Markup Language (SAML) [36] is an Extensible Markup Language (XML)-based standard to transfer identity data between Identity Providers (IdPs) and Service Providers (SPs). The identity data is sent as a *SAML assertion*. This is an XML document signed by the IdP that typically contains attributes about the user. SAML allows Single Sign-On (SSO), i.e. a user can use one set of credentials to log in to multiple services. To enable SSO, services and IdPs need a "trust"-relationship. This trust is established through an *identity federation*. An identity federation is defined as “a group of Identity and Service Providers that sign up to an agreed set of policies for exchanging information about users and resources to enable access to and use of the resources” [18]. Thereby, a federation can consist of multiple sub-federations, e.g. bwIDM is a sub-federation of

the German research and education federation DFN-AAI. DFN-AAI [14] is a SAML federation operated by DFN that enables users from research and education institutions in Germany to access protected resources, such as HPC systems. DFN-AAI also participates in eduGAIN [17] - an inter-federation service that connects identity federations around the world. With eduGAIN nearly 27 millions students, researchers, and educators from over 2800 IdPs can access more than 2000 SPs through a single trusted identity at their home IdP. These numbers show the importance of SAML in today's research and education AAIs. In 2015 Ping Identity stated [39] that "SAML holds the dominant position in terms of industry acceptance for federated identity deployments". However, as previously mentioned there is also a rising trend of AAIs building on OIDC. And while it is clear that SAML is an important technology in many AAIs today, from our experience, SAML is used less than OIDC in new software. Also, SAML was designed to be used within web browsers and it is less suited for usage with native applications. Native applications are non-web browser applications, such as desktop, commandline, or mobile applications.

2.1.4. OAuth 2.0

The OAuth 2.0 (also called OAuth 2, OAuth2, or just OAuth) authorization framework [26] provides mechanism to give access to resources or services, similar as it is possible with SAML. To do so, OAuth uses tokens, namely *access tokens* and *refresh tokens*. We will elaborate on these in subsection 2.2.2. The communication uses JSON over https. Unlike SAML, OAuth also supports mobile and native applications.

2.1.5. OpenID Connect

OIDC [42] is an authentication protocol building on top of OAuth 2.0. OIDC extends OAuth with new functionality, mainly related to authentication. It introduces ID tokens that encode information about the user as well as metadata about the authentication, and also adds the userinfo endpoint which can be queried by applications to obtain user attributes. Based on these attributes SPs can implement more sophisticated authorization policies, e.g. group-based authorization.

2.2. OpenID Connect

From the introduced protocols OIDC is the most recent and modern protocol. It is widely used in modern AAIs. We already mentioned in the introduction that many social identity providers and enterprises (e.g. Google, Microsoft, IBM) build on OIDC. We also mentioned national and international projects and organizations (e.g. EGI, NHS, HIFIS) widely using OIDC in their infrastructures and WLCG moving to token-based authorization. Therefore, we strongly believe that OIDC will remain to be a key technology in modern AAIs for the foreseeable future. For this reason and because this work is heavily connected to OIDC we describe it more detailed in the following. Without claiming completeness we give an overview of OIDC that focuses, but is not limited, on the parts that are particularly relevant for the work of this thesis. Since OIDC and OAuth 2 are strongly related much of

the following is also applicable to OAuth 2. Indeed, the mytoken service, developed as part of this thesis, can also be used with OAuth 2 providers with no or little adaption.

In the following sections, we cover general terminology (subsection 2.2.1), the different token types (subsection 2.2.2), and different OIDC flows (subsection 2.2.3 and subsection 2.2.4). In the end of this section we check how the long-running-jobs problem may be solved with mechanisms provided by OIDC (subsection 2.2.5).

2.2.1. General Terminology

In this section we cover some general terminology which is often used with OIDC. We focus on the different roles / entities. The different tokens and terms related to them are described in the next section (subsection 2.2.2).

OIDC Provider An identity provider that supports OIDC is called OIDC provider or OpenID Provider (OP). This is the service where the user has an account.

OIDC Client Applications that want to obtain tokens from the OP must register themselves as a client with the OP. The client registration (subsection 2.2.4.1) will result in a client configuration including client credentials. The term "OIDC client" often refers to this configuration / credentials, but is also used to refer to the application / service which is a client to the OP.

Relying Party An OAuth 2 client application using OIDC is referred to as Relying Party (RP). Commonly it is also just called OIDC client.

Protected Resource The term Protected Resource (PR) can refer to a resource server (such as a cloud storage) or to a single resource on a server (such as a file). In both cases the resource should not be freely accessible but protected using OIDC.

Authorization Server OAuth 2 defines the term Authorization Server (AS) for the server that issues the tokens after a user successfully authenticated. In OIDC this is the OP.

Resource Owner The resource owner is an entity that is capable of granting access to a PR. This is often the user.

2.2.2. Tokens

As previously noted, OIDC is based on a set of different tokens. In this section we describe these token types and how each is commonly used. All of these tokens are issued by the OP.

2.2.2.1. Access Tokens

Access tokens (commonly abbreviated AT) are the most commonly used tokens in OIDC and OAuth 2. They are short lived credentials used to access PRs. Access tokens are so called *Bearer* tokens [29], i.e. they can be passed around and whoever presents the token

will be authorized (if the token is valid); there is no other authentication performed. While this brings security risks, because an attacker that can obtain an access token may use it, it is very practical for legitimate use cases. This is always the case when a resource server has to contact other protected resources in order to fulfill the request. We will see such an example later in subsection 2.2.3.4. However, the bearer property is also useful in simpler cases where a PR uses the presented access token to obtain user attributes like group membership from the userinfo endpoint (see subsection 2.2.4.3) to decide whether access should be granted or not. The risk of a stolen access token is mitigated by the limited lifetime of access tokens, so that an attacker who can take possession of a token can use it only for a limited time. Usually access tokens are valid for minutes up to a few hours.

Scope All issued access tokens have a scope. This scope is a single string of space delimited values. The scope of a token describes what can be accessed with the token, or more generally what can be done with an access token. Clients can specify the scope values wanted for an access token when requesting the token. In the authorization flows (see subsection 2.2.3) the user will be presented with a consent screen where they authorize the application to get the requested tokens. The scope values of the access token are also present in this consent screen. This allows the user to see what information are released to the application and they can allow or deny this access. On some OPs the user also might be able to change (i.e. deselect) scopes. An example of such a consent screen that displays the requested scopes is given in Figure 2.2.

OAuth 2 does not define specific scope values. This is left to the OPs. Generally, it is also possible that clients can define their own scope values. This allows that a PR like a cloud storage service could only accept access tokens with a scope value of `storage.read` for reading the storage and a scope value of `storage.write` for writing to the storage.

Unlike OAuth 2, OIDC defines some special scope values: The `openid` scope value must be included in authorization requests to actually be an OIDC request. OIDC also defines the special scope `offline_access` that is used to request refresh tokens (see subsection 2.2.2.2 and subsection 2.2.3.3). Furthermore, OIDC defines the following scope values with regard to released user information. These scope values can be used to request the user information claims in an ID token (subsection 2.2.2.3) and returned from the userinfo endpoint (subsection 2.2.4.3).

profile Gives access to basic profile information. Among others the possible claims include information about the user's names, preferred username, profile picture, gender, birthday, and locale.

email Gives access to the user's email address and a claim stating if it was verified or not.

address Gives access to the user's physical address.

phone Gives access to the user's phone number and a claim stating if it was verified or not.

Audiences Audience is a mechanism to restrict *where* an access token can be used. The concept is rather simple, but not specified by OIDC in detail. Indeed, the specification

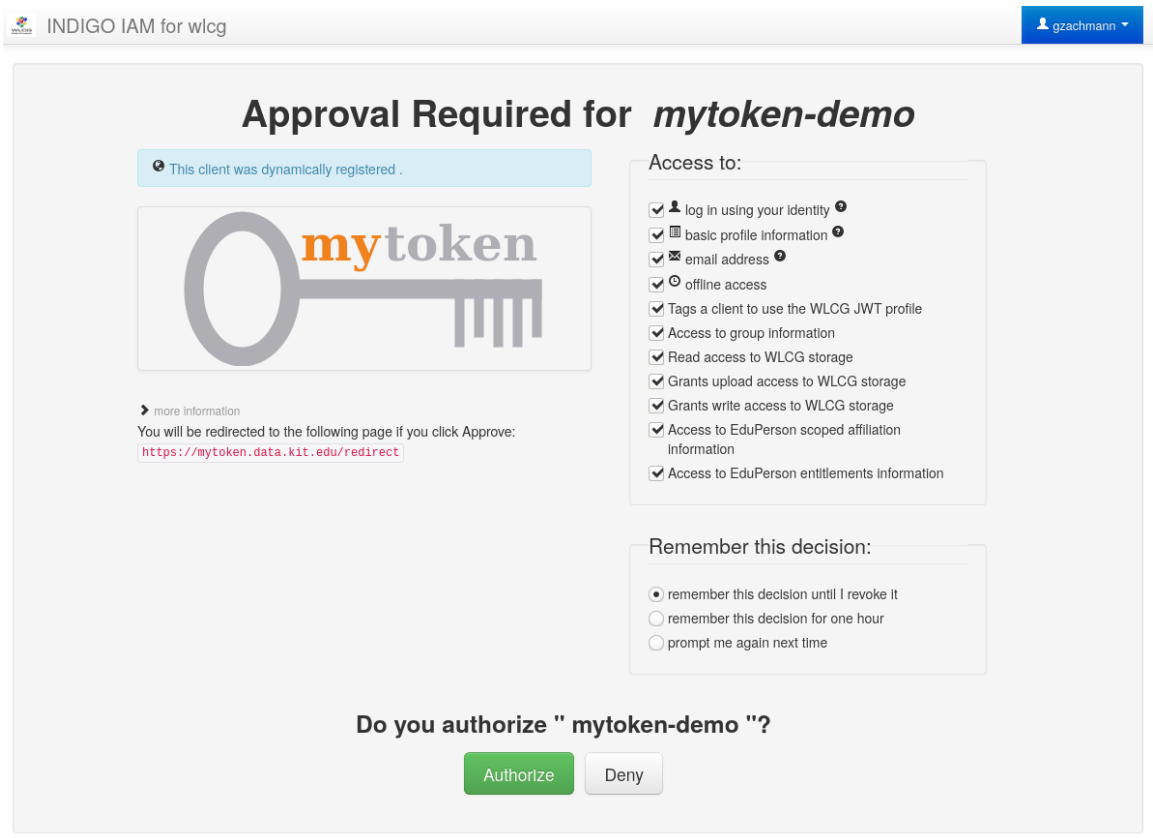


Figure 2.2.: Consent screen of the INDIGO IAM OP [5] for an authorization request

[42] only mentions audience restriction of access tokens very briefly under "Security Considerations" to mitigate the risk that an access token may be used else where than intended. While the OIDC specification does leave audience restriction very open, the idea becomes clear from the brief description: An access token includes an identifier for the resource where it is intended to be used and the resource verifies that it's identifier is included in the audiences of the presented token.

While the OIDC specification only has that one small paragraph about audience restricted access tokens, it does not omit audience restriction completely. The audience restriction is specified for ID tokens (subsection 2.2.2.3) and other objects that are represented as a JSON Web Token (JWT). The JWT specification [28] defines the aud claim used to specify the audience(s) of a token as a standard claim for JWTs. So OIDC mandates the audience claim in ID tokens, but not in access tokens, because the latter are defined as opaque. However, because the concept of audience is defined by [28] and therefore should be known to entities handling JWTs, protected resources should be able to implement audience restrictions.

In practice audience restricted access tokens are currently not widely used with OIDC and one problem is that the OIDC specification does not describe how clients can request access tokens with specific audiences. The lacking support of audience restrictions may be due to the missing specification on how to request such restrictions. However, there is a recent draft [3] for using JWTs with OAuth 2 that describes how to request audience

restriction for such tokens. We hope that when this draft becomes an RFC OPs, it finds adaption and audience restrictions of OIDC access token will finally be more widely used.

2.2.2.2. Refresh Tokens

As already stated, access tokens are short-lived and will eventually expire. However, there are many valid use cases where clients need to access resources over a longer period than the lifetime of a single access token; one example is the long-running-jobs problem. Whenever such an extended access is needed, multiple access tokens must be used, which implies that the client needs to be able to obtain fresh access tokens. The simplest approach would be to let the user perform the authorization flow again. However, this is often not possible because the user is not present anymore. Even when it would be possible, prompting users every hour to log-in again (at the OP) is obviously not user-friendly.

Refresh tokens (commonly abbreviated RTs) are used to obtain new access tokens after the initial (or previous) expired. This can be done by a client using the refresh flow (subsection 2.2.3.3). The refresh token is a much longer lived token that can be valid forever, i.e. it does not expire at all. However, security policies can limit the lifetime of such long-lived credentials, so that OPs might limit the lifetime in practice. However, the lifetime is usually still in the range of months to a few years and therefore can be used to extend the lifetime of access tokens over a long period of time.

Generally, from a security-perspective it is undesirable to have such a long-living credential without any restrictions. Otherwise, access tokens could just be long-lived. Therefore, refresh tokens are restricted in two ways:

First, refresh tokens are not automatically issued to clients; a client must request a refresh token explicitly. This is usually[1] done trough the offline_access scope. Additionally, the user will be informed about this request and has to give their consent (as it is the case for all requested information). An example for such a consent screen that also includes the offline_access scope was already given in Figure 2.2.

Second, unlike access tokens, refresh tokens cannot be passed around, because each refresh token is bound to the client that requested it. This means that a stolen refresh token cannot be used without the client credentials. Nevertheless, it is important that clients handle refresh tokens with care and store them securely.

2.2.2.3. ID Tokens

The ID token was introduced with OIDC and is not known in OAuth 2. In OIDC a client receives an ID token along with the access token after an authorization flow. The ID token is a JSON Web Token (JWT) that encodes information about the user. The client can decode the signed JWT and use the user information, e.g. for personalizing the webpage with the user's name, but also to obtain the user's email or physical address. The content of the ID token is generally limited by the requested scopes, but the application can limit it further by requesting only the needed claims.

[1] Google instead uses the access_type=offline request parameter.

2.2.3. Important Authorization Flows

OIDC specifies several so called Auhtorization Flows for obtaining the introduced tokens. In the following we describe the most important of these flows. All of the following flows have the goal to obtain an access token (the token exchange flow (subsection 2.2.3.4) can also be used to obtain other tokens, but we will focus on access tokens). However, only the first two (authorization code flow (subsection 2.2.3.1) and device flow (subsection 2.2.3.2)) require user interaction. One could say that these flows start with "nothing", i.e. a client does not already have some sort of credentials for the user, and therefore require the user to log-in with the OP. The other two flows (refresh flow (subsection 2.2.3.3) and token exchange flow (subsection 2.2.3.4)) do not involve the user, i.e. they can be done by the client without the user, but require an already existing credential for that user.

2.2.3.1. Authorization Code Flow

The authorization code flow is considered the standard OIDC/OAuth 2 flow. It certainly is the default flow on the web and therefore, also the most used and best supported authorization flow.

We want to note that the user does not give credentials to the application (the OIDC client), but only logs-in with the OP. This is true for most authorization flows and not only for the authorization code flow, but we want to stress this important property at this point.

When a user starts the flow the client will construct an authorization URL and redirect the user to it. This will redirect the user to the OP. There the user has to log-in. The OP displays a consent screen (see Figure 2.2) that details the released information. The user can then decline the request or approve it. After approval the OP makes an https redirect to an endpoint of the application (which was registered beforehand). This redirect will also contain an *authorization code*. This code is the name-giving component of this flow. The application uses the authorization code to exchange it into an access token and possibly also a refresh token. With OIDC the application will also receive an ID token. The authorization code flow is visualized in Figure 2.3.

2.2.3.2. Device Code Flow

While the authorization code flow works fine on the web, it is less suitable for native applications. The device code flow [13] is an authorization flow specially for input-constrained devices and devices without a web browser (but with an internet connection). An example for an input constrained device is a TV, that might have a web browser, but it usually does not have a keyboard and using the remote to navigate on an on-screen-keyboard to enter a (long and complicated) password is intricately. Therefore, the device flow supports these devices, by letting a user log-in through another device.

The device code flow is started on the primary, constrained device. The application receives a *user code*, *device code*, and an authorization URL from the OP. The device code is only for the application; it is used for polling the OP for the issued tokens. The user code and authorization URL are presented to the user. An application can visualize the

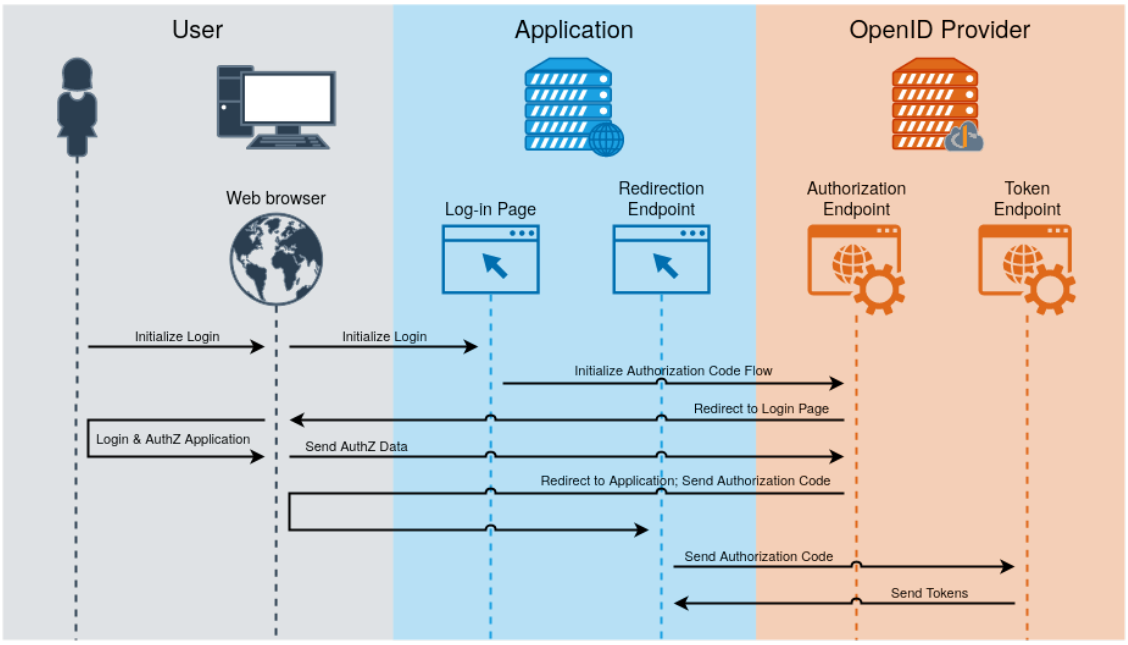


Figure 2.3.: Visualization of the Authorization Code Flow

information by simply printing it on screen, or by rendering it as a QR code, so that the URL can easily be opened with a smartphone, for example.

The user opens the authorization URL on another device, the smartpohne, in our example. This device should now have a web browser and better input capabilities. The user logs-in with the OP, enters the user code, and authorizes the application in the consent screen. The user no longer needs the second device and can now return to the primary one.

As stated above, the application started polling the OP for the tokens, in the meantime. Until the user approves the request on the second device the application does not receive any tokens from the OP; instead it receives a message indicating that the authorization is still pending (if the user declines the request, the OP answers with an appropriate message). Eventually the user authorized the application and the application receives an access token and possibly a refresh token as well as an ID token in case of OIDC from the OP. The device code flow is visualized in Figure 2.4.

2.2.3.3. Refresh Flow

We already referred to the refresh flow when explaining refresh tokens. As stated, the refresh flow is a rather simple flow to obtain a fresh access token from a refresh token. Due to the short lifetime of access tokens, applications need a way to obtain new access tokens after one expired. We already explained that applications can obtain refresh tokens with the previously explained flows by including the offline_access scope in the request. In the refresh flow this refresh token is presented to the OP in order to obtain a new access token without the user having to be present. This simple flow is also visualized in Figure 2.5a.

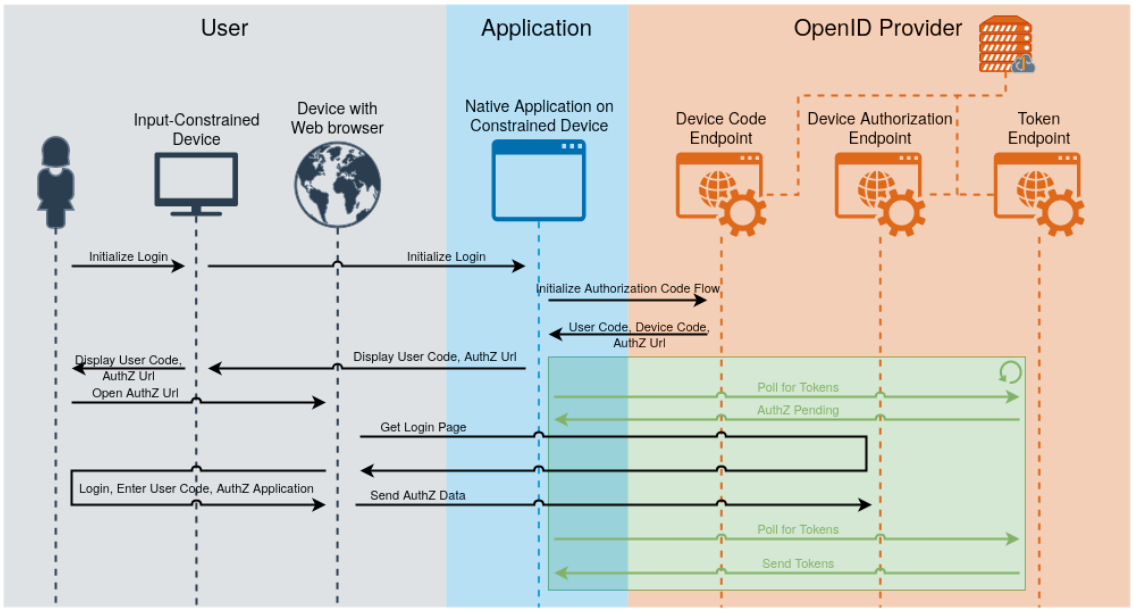
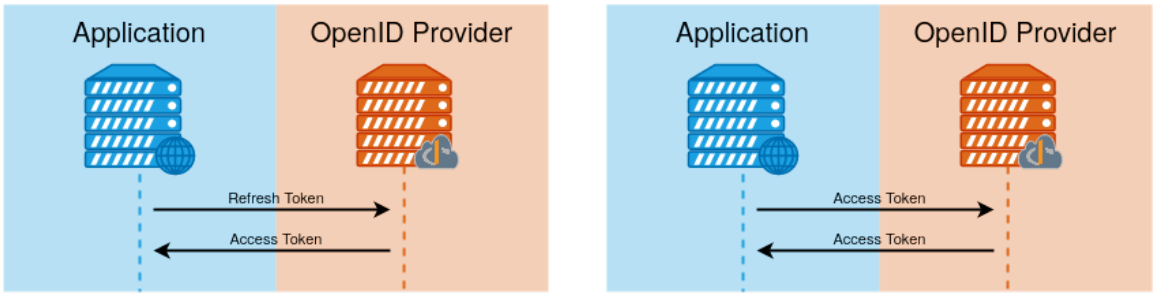


Figure 2.4.: Visualization of the Device Code Flow



(a) Visualization of the Refresh Flow

(b) Visualization of the Token Exchange Flow

Figure 2.5.: Visualization of the Refresh Flow and Token Exchange Flow

2.2.3.4. Token Exchange Flow

The token exchange flow [30] can be used to exchange one token for another. The principle of the flow is visualized in Figure 2.5b. When comparing this figure with the one for the refresh flow, they look very similar. However, this is only the case because Figure 2.5b is simplified and omits other paramters and variations of the flow. In reality the token exchange flow is much more complex, flexible, and also powerful.

In the following we motivate the token exchange flow. Assume the following scenario, which is visualized in Figure 2.6: A user U uses a client C which needs to access the protected resource R_1. C therefore uses an access token t_1 of U to act on their behalf. This is the normal OAuth 2 / OIDC use case.
However, it might be that resource R_1 must access another resource R_2 in order to fulfill the request. Because the access token t_1 that was presented by C to R_1 is a bearer token and can be passed around, R_1 could use this token to access R_2. Usually, this succeeds and is not a problem. However, there are situations, where there still is an issue:

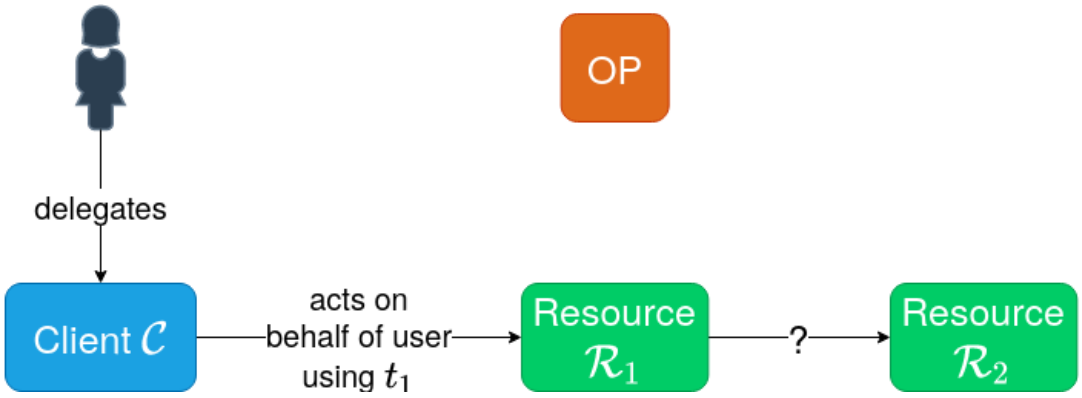


Figure 2.6.: Use case for Token Exchange [6]

a) R_2 needs additional / other scopes than R_1

b) The access token t_1 is audience restricted

In both situations the token t_1 cannot to be used by R_1 to access R_2. In a) because the required scopes are missing, in b) because the token is only meant to be used at R_1 and R_2 therefore should not accept this token. In theory, both cases can be easily solved if C sends an access token that includes all the needed scopes and audiences. However, in practice this might not be feasible, because it would require C to know which resources R_1 will access and which scopes are required. This additional information is unknown to C.

However, the token exchange flow allows to exchange one token for another. In particular it allows to exchange an access token with some scopes s_1 and the audiences a_1 for another token with scopes s_2 and audiences a_2 without the need that $s_1 \supseteq s_2$ and $a_1 \supseteq a_2$.

In our example this allows C to send the access token t_1 with only the scopes and audience needed for R_1. R_1 then can exchange the token t_1 for t_2 with R_2 as audience and the correct scopes, and then use t_2 with R_2.

It can be seen that token exchange is a powerful tool in such use cases. However, it also becomes clear that this power comes with risks. The token exchange flow presented allows a client to work around the scope and audience mechanism that were introduced for good reasons. At this point we want to make clear, that token exchange can also be used to exchange an access token for a refresh token. This also does have its legitimate use case, but brings some risks, because it allows turning any valid access token into a refresh token, which renders the limited lifetime of the access token effectively useless (for an attacker that is able to perform token exchange). We also want to point out, that all of this is done without user interaction and without notice of the user.

Therefore, we recommend that OPs should make sure that only trusted clients are allowed to use the token exchange flow[2], and even then should implement mechanisms to restrict this further, e.g. a client could be allowed to obtain only access tokens (no refresh tokens) and only with certain scopes and audiences[3]. This way, the client can still

[2]This is currently the state of the art for OPs that implement token exchange.
[3]We are unaware of any OP that already implements such a feature.

use token exchange for legitimate cases, but it is not possible to exchange any severely restricted access token for a basically unrestricted token.

2.2.4. Auxiliary Flows and Endpoints

In the following we present some additional flows, namely client registration and token revocation, as well as the userinfo endpoint.

2.2.4.1. Client Registration

A client has to register itself with the OP. As part of the registration process it receives its client credentials which must be used as authorization in most requests to the OP. Any application that wants to receive tokens directly from the OP needs client credentials and therefore must register an OIDC client. A PR that only receives access tokens as part of a request and does not obtain tokens directly from the OP may not have to register a client. However, this depends on the authorization policy that is in place, the information needed to evaluate the authorization rule, and where this information can be obtained. The last part refers to the set of endpoints of an OP which offer different pieces of information. If a PR needs to access information that is not available from the token itself or from the userinfo endpoint (subsection 2.2.4.3), but only from other endpoints (e.g. the token introspection endpoint), the RP must be authorized and therefore needs to register itself with the OP. In such a case some OPs might allow a resource to register itself as a PR and not as a client, while other OPs do not make this distinction. In the following we will focus on client registration, however, the information also applies to registering a PR, respectively.

There are two options for a client to register itself with the OP: registration through a web interface and dynamic client registration. An OP might support only one of these, or both. The OP could also support none of these and do the registration out-of-band, e.g. via email; however, for a production OP this is generally not feasible.

Registration through a Web Interface Registering a client through a web interface feels natural for many users[4] and generally is much more user-friendly than dynamic client registration. The user interface can help with the registration process and give useful background information.

In practice the complexity of the web interfaces of the different OPs varies. Some OPs have a rather clean interface with less options while others need a more complex interface because they offer more configuration options. Google[5], for example, offers very limited configuration options, while the INDIGO IAM[6] instances offer much more options, like choosing grant types (i.e. the available authorization flows) or the cryptographic algorithms used for different operations, like token signing. EGI Check-in[7] even supports

[4] Note, that users in this case are usually administrators, so that this statement is somewhat put into perspective.
[5] https://accounts.google.com/
[6] There are multiple instances for different community deployed, e.g. https://wlcg.cloud.cnaf.infn.it/
[7] https://aai.egi.eu/oidc/

configuring different properties of the issued tokens, like the lifetime for each of the different token types and whether refresh tokens are rotated after usage or not.

All web interfaces we are aware off, require an authenticated user; so in terms of dynamic client registration which will be explained in the next paragraph, this is a *protected* client registration.

Depending on the OP's policy a client might be immediately registered and activated or not. Some OPs may require approval of client registration requests by an administrator or they require that a policy document is filled in.

Dynamic Client Registration While the registration via a web interface generally feels more user-friendly, it is not suitable to register clients *dynamically*, i.e. non-interactively. With the dynamic client registration [41] applications can register themselves as a client without human intervention. (This is only true for the registration itself as it is performed by the client application; if the registration has to be approved on the OP side by an administrator or an additional policy document has to be filled in, human interaction is still required.)

To register a client the application sends the requested client configuration as a JSON document to the OP's registration endpoint. The OP processes the request and upon success sends back an updated client configuration, which includes the client credentials. It usually also includes a registration access token that can be used to update or delete the client.

For applications this flow is much more suited than registration via a web interface. The application can use a predefined client configuration (and adapt it, if necessary) to send it to the registration endpoint.

Dynamic client registration can be *open* or *protected*. Open dynamic client registration does not require any authorization, i.e. anyone can send a client configuration to register a client. This can be useful for applications that register themselves dynamically as a client, because they do not need any additional information; however, often security policies do not allow this form of client registration.

Protected dynamic client registration on the other hand does require some sort of authorization. The exact means of that authorization are not specified clearly. But it will involve an initial access token that is provisioned out-of-band. This initial access token could be a dedicated token that can only be used to register one or multiple clients, but it could also be a normal user access token. The client registration can then still be limited to certain users / users of a certain group.

2.2.4.2. Token Revocation

The revocation of tokens [32] is an important security feature. Because of their long lifetime it is strongly recommended to make refresh tokens revocable, so that users can revoke a refresh token when it gets stolen. The RFC specifying token revocation [32] is an additional specification (not included in the OIDC core) and the OP might not implement it. When implemented, it mandates that refresh token must be revocable and that access tokens also should be revocable.

2.2.4.3. Userinfo Endpoint

The concept of the userinfo endpoint is simple: It is an endpoint of the OP where user information can be obtained. It is actually a PR and requests to it must include an access token for authorization. Then the userinfo endpoint responses with information about the user associated with the presented token. The information returned depends on the request and the scopes of the token. We detailed scopes defined by OIDC for the userinfo endpoint and which information they give access to in section 2.2.2.1. An example response can be seen in Listing A.1.

2.2.5. Mechanisms to Solve the Long-running Jobs Problem

OIDC provides some mechanisms which can be used to overcome the lifetime limitation of access tokens. In the following we describe how to extend the lifetime of access tokens with only OIDC mechanisms and note why these mechanisms lack usability.

2.2.5.1. Refresh Tokens

Refresh tokens are the obvious way to extend the lifetime of access tokens. In the following we shortly recap the most important properties:
A client can obtain a refresh token when the user performs one of the authorization flows and the client includes the offline_access scope. By its nature the refresh token is long-lived and potentially does not expire at all. Therefore, it is a powerful token. Unlike access tokens, refresh tokens should not be passed to other entities but are only meant for the client that requested them. In order to enforce this, but also to limit the risk of a stolen refresh token, refresh tokens are bound to the client to which they were issued. When a client wants to use a refresh token it must authenticate using its client credentials.

These properties make refresh tokens less useful for the long-running-jobs problem. Different approaches to use refresh tokens with the long-running-jobs problem are possible. In the following we will describe two and their limitations:

Start Job with a Refresh Token Instead of using an OIDC access token as authorization to start a compute job, a refresh token is used. This allows to exchange the refresh token into an access token whenever an access token is needed. However, since refresh tokens are bound to the client they where issued to, one does not only have to pass the refresh token but also the client credentials. It is easily doable to encode all the needed information into a single token string.

One must consider that a user now needs access to a registered OIDC client. Again there are two possible solutions: Multiple users can share a client or each user has its own client.

If multiple users share a client (i.e. they all know the client credentials), this strongly increases the risk that a stolen refresh token can be abused. Letting multiple users share a client in order to be able to pass refresh tokens around defeats the purpose why refresh tokens are restricted to one client.

If each user has its own client, it must be registered. If the OP supports dynamic client registration [41] (section 2.2.4.1) this can be easily done. If the OP does not support dynamic client registration the client has to be registered manually by the user, which is never ideal and even less if it is an average user without any detailed knowledge about OIDC.

No matter how the user has access to a registered client, after the job finished, the used refresh token should no longer be valid. Most likely the user does not have control over the lifetime of the refresh token (see subsection 2.2.5.3 for a variant where the user can control the lifetime of OIDC tokens) which means that the token will still be valid for a long time (up to infinity) after the compute job ends. Therefore, the refresh token should be revoked after the job ends. However, this means that each time a compute job is started the user needs a new refresh token. Because refresh tokens can only be obtained through an authorization flow[8] this requires true user interaction, i.e. the user has to login at the OP which cannot be done completely on the command line. Furthermore, this approach will most likely not be compatible with security policies of existing infrastructures.

Start Job with Authorization Flow Another way to use refresh tokens within a long running job is to use a central client for all users. This means that this client is confidential and none of the users knows its credentials. This approach is right inline with how applications usually handle refresh tokens. Indeed, the user never sees the refresh token and it is only known to the job submission service (the OIDC client in this case). This means that the user does not authorize job submission with a token, but directly with the authorization flow, e.g. the device flow. Similar to the previous approach a user has to do an authorization flow for each started job. Apart from that this approach is a clean solution to the problem. However, it requires substantial adaption of the job submission service, in particular it must implement a mechanism allowing jobs to always obtain a valid access token.

2.2.5.2. Token Exchange

The token exchange flow [30] (subsection 2.2.3.4) can be used to exchange one token into another. It is even possible to exchange a token into a higher privileged token. Among others it is possible to exchange any access token into a refresh token which then can be used to obtain additional access tokens. While this can be used to solve the long-running-jobs problem there are still some caveats. Both the token exchange flow and the refresh flow require a registered OIDC client. However, in this approach it is not an option to have a client per user, so the fact that there is need for a registered client is not an issue. Rather, the problem is that the client needs a special permission in order to perform the token exchange flow. Because the token exchange flow is very powerful it is important that OPs are cautious with enabling the token exchange flow for clients. However, we can assume that it is not a problem to register a client with the OP for this specific use case.

[8] The exception is token exchange on which we comment in subsection 2.2.5.2.

The main problem with this solution is the lack of support by the OPs. While there are OPs that support the token exchange flow (INDIGO IAM, EGI Check-in), it is still not widely supported.

Similar to the job submission service with a client that only does the refresh flow, a service that additionally does token exchange also needs significant adaption as well as a way to provide access tokens to the computing jobs.

2.2.5.3. Dynamic Client Configuration

This approach requires each user to have it own registered OIDC client. However, because dynamic client registration is used this should be transparent to the user. If an OP supports dynamic client registration that can also be used to adapt the client settings after registration and the OP allows a client to set the lifetime of access tokens for this client, these can be combined to create access tokens with a user-defined lifetime. The idea is the following:

1. Dynamically register a client.
2. User performs once an initial authorization flow to obtain a refresh token.
3. User requests an access token with lifetime t_1.
4. The client adapts its settings, i.e. it sets the lifetime of access tokens issued to it to t_1.
5. The client performs a refresh flow with the refresh token obtained in step 2. The OP will issue an access token with the lifetime t_1.

Steps $1 - 2$ only have do be done once per user. Afterward steps $3 - 5$ can be done as often desired.

Each time the user requests an access token with a specific lifetime the client's settings are adapted to be in line with that request. While this adds an overhead (it adds an additional request, which doubles the number of requests), it is most likely negligible. The bigger problem is that setting the lifetime of access tokens for a client is a feature that is barely supported by OPs. At the time of writing we are only aware of one OP (EGI Check-in) that does support this feature.

At this point we want to note, that it would be desirable to directly request access tokens with a specific lifetime from the OP. This would elegantly solve the long-running-jobs problem because then one can request an access token that is just valid long enough for the job to complete. However, to our knowledge there is no OP implementing this feature. Furthermore, it is also a security sensitive topic, since access tokens are easy to use (and can be passed around) without any other restrictions.

2.3. Related Work

While there are mechanisms provided by OIDC to extend the lifetime of access tokens, they are all not suitable to solve the long-running-jobs problem, mainly because of a lack

of OP-support, user-friendliness, or substantial adaption effort. In this section we look at three tools that can be used to obtain access tokens on the command line and could also be used to solve the long-running-jobs problem.

2.3.1. EGI FedCloud

A solution of the long-running-jobs problem was needed for development on the EGI Federated Cloud. For this, the EGI-AAI provided a simple web service [19], to which the user logs-in via EGI Check-in, so that a refresh token using the authorization code flow is obtained. Then the web page displays the refresh token, the client credentials, and a curl command to the user. The curl command can be used on any machine to obtain access tokens. It includes the refresh token as well as the client credentials which are needed for the refresh flow. The command looks like the following:

```
curl -X POST -u 'f33e824a-078d-497b-b700-25b0df7fc5b7':'B80vPK0LVbYuvwRj0
↪ Aexs8y0rKgk5XHwYRRq3BCr33ejj33385bzDVcPmSTUkqA2QjMiwWKJDT
↪ xvOou7yVV8EA'  -d
↪ 'client_id=f33e824a-078d-497b-b700-25b0df7fc5b7&client_secret=B80vPK0L
↪ VbYuvwRj0Aexs8y0rKgk5XHwYRRq3BCr33ejj33385bzDVcPmSTUkqA2Qj
↪ MiwWKJDTxvOou7yVV8EA&grant_type=refresh_token&refresh_token=
↪ <refresh_token>&scope=openid%20email%20profile'
↪ 'https://aai.egi.eu/oidc/token' | python -m json.tool;
```

We replaced the refresh token with a placeholder in the above command, but did not replace the client credentials, because they are the same for all users. Consequently, they are not kept confidential. However, client credentials must be kept confidential according to the specifiction[9]. The same is true for refresh tokens. This becomes even more important when the client credentials are public.

As indicated, this simple web service enables users to use the displayed curl command on any device to obtain access tokens. However, the long command is not user-friendly, especially if a user wants to change the requested scopes. Furthermore, the command contains sensitive information. On a shared system this information can be leaked through the bash history or through a ps call. An attacker then can use the stolen refresh token together with the client credentials to obtain access tokens and ultimately impersonate the user.

In summary, the simple service provided for EGI FedCloud, enables users to obtain access tokens through a curl command. However, the user experience is mediocre and, most importantly, the service has several severe security flaws and therefore will be removed soon, after our intervention.

2.3.2. oidc-agent

The oidc-agent [47, 48] was developed at KIT and is a tool for obtaining OIDC access tokens on the command line. This goal is very similar to the one of this thesis. However,

[9] Unless it is a public client, in which case most OPs (including EGI Check-in) do not issue a client secret at all. Because this client has a client secret, it is not a public client.

it has one big drawback: it is not a central service and is therefore much less suited to obtain access tokens from multiple machines.

The oidc-agent is a tool-suite that consists of multiple components:

oidc-agent runs as a daemon and handles all communication with the OP. It caches and refreshes access tokens when needed. Other applications can always obtain a valid access token from the agent.

oidc-gen is used to create *account configuration files*. These files contain the OIDC client credentials and a refresh token. The file is stored in an encrypted way on the user's system's disk.

oidc-add can load and unload the account configuration to the oidc-agent. It is only possible to obtain access tokens for loaded accounts, but it is also possible to load an account configuration on-the-fly, meaning that it does not have to be loaded beforehand with oidc-add.

oidc-token can be used to obtain an access token from the agent. The token can then be copied to use it elsewhere or the call to oidc-token can directly be integrated in another program call.

liboidc-agent The project also provides libraries for the C, Go, and Python programming languages. These allows developers to integrate their applications with the agent and obtain access tokens directly from the agent.

oidc-agent was specifically developed as a *local* service that runs on the user's machine. While this gives the user full control over their data, it also has some disadvantages:

Client Registration As stated in subsection 2.1.5 client registration support greatly differs between OPs. For OPs where users must register their own client manually them-self, the client registration can be an obstacle.

Other machines Usually a user installs the agent on their machine where all account configurations are stored. But, obtaining access tokens from multiple machines is - generally - out of scope. oidc-agent has support for agent forwarding with ssh [46], but this is limited and not enough in many use cases and our requirements.

While oidc-agent offers the needed functionality on one local system, it is not suitable as a tool for obtaining access tokens easily on any machine. When creating an account configuration it is possible to restrict the scope and audience values, but it is not possible to restrict the lifetime. The account configuration is essentially an OIDC refresh token with the used client credentials and therefore, has the same lifetime as the refresh token.

2.3.3. htgettoken

The htgettoken [16, 15] tool was developed at FermiLab and is a central token service similar to the envisioned mytoken service. htgettoken builds around the well established

secret managment software vault by Hashicorp [27]. The project provides RPM packages to extend and configure a vault server with the required plugins and currently unmerged pull requests. Also a commandline client called htgettoken is provided. This client communicates with the vault server and can be used to obtain OIDC access tokens.

The htgettoken tool obtains access tokens through vault by authenticating with a vault token. If htgettoken cannot find a vault token, it tries to retrieve a vault token using Kerberos authentication, and if that does not succeed, an OIDC flow is started. At the end of the authorization flow the vault server obtains the refresh token, stores it, and returns a vault token. The integration with Kerberos was important for htgettoken and brings good usability for these users.

The ideas for htgettoken and the mytoken service were developed in parallel. htgettoken builds on extending well-established software while mytoken starts with a new implementation to be more flexible and not constrained by existing software. While at the beginning of the project it seemed like htgettoken would not support requesting access tokens with specific *scopes* and *audience* values, this is the case and htgettoken supports these features. It is also possible to configure the lifetime of the vault token.

By default htgettoken stores the vault token without any encryption on the user's machine under /tmp/vt_u\$uid. It is also possible to print the vault token to another file or to stdout. If a vault token with a lifetime greater than one million seconds (ca. 11.5 days) is requested, this token cannot be stored in a regular file, but can only be printed to files under /dev, e.g. stdout. This behavior reflects policies, stating that user credentials should not be stored without encryption for a longer time period.

As just indicated, htgettoken stores the vault token unencrypted on the user's machine. While Kerberos users can use faster-expiring vault tokens and obtain new vault tokens through Kerberos in a user-friendly way, this is not possible for users without Kerberos. We see three options for non-Kerberos users:

- Use vault tokens with a rather short lifetime (much shorter than the maximum), so that the security risk of an unencrypted vault token is acceptable. However, this requires the user to log in often through OIDC which lowers usability significantly.
- Use vault tokens with a rather long lifetime, so that the number of re-authentications through OIDC can be limited. However, this increases the risk that a vault token can be misused.
- Store vault tokens in an encrypted way, so that the security risk is minimized. This is certainly the preferred way, but it requires that users manage the encryption/decryption of vault tokens in some way, which reduces usability.

We will compare the mytoken service we developed to htgettoken and oidc-agent in section 5.2. We do not consider EGI FedCloud any further because of its security flaws.

3. Design

In this chapter we present the design of the mytoken service we developed. After presenting the requirements, we describe the key concepts and endpoints of the mytoken service.

3.1. Requirements

In this section we describe the requirements for the mytoken service that should be developed.

The two main requirements are:

Command line usage The primary use case involves the command line. It should be easy to obtain access tokens on the command line. The clear focus is on command line usage; a web-interface for the mytoken service would be nice to have, but is optional and out of scope of this thesis.

Machine independence It should be easy to obtain access tokens on any machine, for example on a remote server or from a compute job.

These two requirements describe the basic idea behind the mytoken service: It should be easy to obtain access tokens on the command line from any machine. Additionally, the following requirements should be fulfilled by the mytoken service:

Security It is clear that this service is a security-sensitive application and attention should be put on this topic.

Provider support The service should be a general service that can support different OpenID Providers (OPs). This means that it should not be a service specifically tailored to work with one OP, but a service that works with any compliant OP.

Multiple providers per instance In addition to the previous point, a single instance of mytoken should be able to support multiple OPs.

Configurability The service should be configurable by both administrators and users:

- **Deployment** Administrators should be able to configure their deployed instance according to their needs. In particular administrators are free to enable and disable certain features and set security and privacy options according to their policy.
- **User** Users should be under full control of their tokens and how to create them.

User-friendliness The service should be easy to use. This includes:

- Easy-to-use and clear interfaces
- Sensible default settings

Extensibility The service should be easily extensible, in particular it should be possible to support additional ways to obtain tokens from the service.

Performance While performance is not a major factor, the mytoken service should still be capable of handling a fair amount of traffic.

High Availability While it was not a requirement to fully implement high availability, the need for high availability should be considered and the service should be designed in a way that it can easily support high availability.

Permissive open source license All source code should be publicly available under a permissive open source license.

Self-hostable Communities should be enabled to host their own instance of the service. This also includes the relevant documentation.

3.2. Concepts

The central challenge that mytoken addresses is to provide a way to solve the long-running-jobs problem in a way that security and usability both can be modeled in a sensible way. The following concepts, especially the restrictions, are central for how mytoken addresses security and usability.

3.2.1. Mytokens

From the requirements (section 3.1) it becomes clear that a user should be able to obtain access tokens with a single token-like string. This is due to the prerequisite that users should easily obtain access tokens from any machine. We call this single token-like string a *Mytoken* and it is the central element of the mytoken service.

The mytoken is a new token type we introduce with the mytoken service. It is only relevant to the mytoken service, i.e. it is only issued by mytoken instances and can only be used there.

The mytoken is a signed JSON Web Token (JWT) that encodes the following attributes:

iss The URL of the entity that issued this token, i.e. the URL of the mytoken instance.

sub A string uniquely[1] identifying the entity this token is issued to, i.e. the subject of the token. The subject string is defined as <oidc_sub>@<oidc_iss>.

exp The UNIX time stamp when this token expires. If the token does not expire exp is 0. Mytoken computes the value from the restrictions attribute.

[1] Unique within a mytoken instance.

nbf The UNIX time stamp when this token can be used for the first time. Mytoken computes the value from the restrictions attribute. A token can never be used before it was issued (iat).

iat The UNIX time stamp when this token was issued.

jti A random and unique identifier for this token. Mytoken uses Universally Unique Identifiers (UUIDs) for this purpose.

seq_no A sequence number. This is currently unused, but can be used in future work (section 6.2) for rotating mytokens.

aud A URL identifying the audience of this token, i.e. where the token is indented to be used. This is the URL of the mytoken instance.

oidc_sub The subject (sub) claim that is returned from the OP about the user for whom this token is created.

oidc_iss The issuer URL (iss) of the OP from whom access tokens should be obtained.

restrictions A JSON array of restrictions (see subsection 3.2.3) for this token.

capabilities A JSON array of capabilities (see subsection 3.2.2) for this token.

subtoken_capabilities A JSON array of capabilities (see subsection 3.2.2). If set, other mytokens created from this mytoken can only have these capabilities. (If not set other mytokens created from this mytoken can only have the same capabilities as the original token.)

Since a mytoken is a signed JWT that encodes all of this information, it is self-contained and can be verified with only the token. This means that any entity can check the content of the token and see what they can do with this exact token while a mytoken instance can verify that a mytoken can be used for a certain action. However, for full verification mytoken still needs to access the database to check that the token has not been revoked and to verify some of the restrictions (i.e. how often the token was used previously).

While a self-contained mytoken can be useful for both users and the mytoken service, the payload of the token is substantial. If multiple restriction clauses (see subsection 3.2.3) are used, mytokens can become quite long. The length of a mytoken can be a problem when it should be used at places where the input length is constrained. Web servers for example commonly limit the size of request headers they accept. The apache and nginx web servers for example have a header request limit of 8 KB by default [1, 35]. Of course this can be adjusted, so it is possible for web servers to accept less (or more) bytes. However, usually the request header limit of web servers is not a problem, but there might be cases where a problem can arise from the limit. A much more relevant problem are input fields that have a much smaller limit, like password fields. One can think of different scenarios where a token should be passed via a password field. For example there are efforts in the EOSC-synergy project to integrate HPC resources with OpenID Connect (OIDC) by using OIDC tokens with ssh [4]. Such a token would be passed as a password. In this scenario a

token larger than 1 KB is a much bigger problem than with https requests. Furthermore, there might be other use cases with much bigger input constrained, i.e. allowing much less characters.

As stated, the length of a mytoken is directly proportional to its payload, but the token length also depends on the signature length which in turn depends on the signature algorithm. An RSA based signature using a 4096 bit key, also has a signature length of 4096 bit, which equals 512 bytes without any encoding. However, mytoken supports different signature algorithms. But even for elliptic-curve based signature algorithms that produce rather short signatures, the total length of a mytoken can easily exceed 1024 bytes.

Therefore, we decided to have different token types for mytokens. Normally a mytoken is a signed JWT which is quite large. But a mytoken can also be returned either as a *short token* or as a *transfer code*.

Short Tokens A short token behaves exactly like a normal mytoken. It can be used everywhere where a JWT mytoken can be used. For all matters, a short token is the same as a normal mytoken. However, the only difference is that a short token is not a signed JWT but an opaque random string that is much shorter. The length of a short token can be configured by the administrator of the mytoken instance and defaults to 64 characters. Internally a short token is mapped to a JWT mytoken.

Transfer Codes A transfer code is an even shorter string (default 8 characters) that can be used to transfer a mytoken from one machine to another or at places that have input constrains, so that even a short mytoken would be too long. A transfer code cannot be used in the same way as a normal mytoken or a short mytoken. It can only be used for exchange into a mytoken. The transfer code is a one-time code, i.e. it can only be used once, and is only valid for a limited amount of time (default is 5 minutes). For example it can be used to transfer a mytoken that was created on one machine to another machine when other ways of transferring (e.g. copy-pasting) are not suitable. Because the transfer code is so short it can easily be typed without the need to copy it.

In mytoken we introduced the mytoken as the only new token (with the three internal types that we just described), but this token can be used for different actions using capabilities which we explain in the next section.

3.2.2. Capabilities

Mytoken *capabilities* are similar to OIDC scopes because they define what can be done with a token. Therefore, capabilities restrict the power of a mytoken. This is true, because only well defined actions related to capabilities can be done with that token. However, the capabilities do not restrict the token in the same sense as the *restrictions*, this becomes clear when looking at the default behavior: A mytoken without any *restrictions* defined is unrestricted and can be used "as normal". A mytoken without any *capabilities* defined is useless because it cannot be used for any action (all actions require some sort of capability).

We want to point out that - because it does not make sense - there is actually no way to obtain a mytoken without a capability.

In the following we list the defined capabilities and the related actions:

AT This capability allows to obtain an OIDC access token from the mytoken access token endpoint. Since the primary focus of the mytoken service is to provide access tokens to applications this is the standard action. Therefore, the AT capability is the default value when a user does not request specific capabilities.

create_mytoken This capability allows the creation of new mytokens from an existing mytoken. This allows users to have long living mytoken with that capability that can be used to create additional more restricted mytokens that then can be passed to other applications when they need to obtain access tokens.

tokeninfo_introspect This capability allows querying the mytoken tokeninfo endpoint for basic information about the token. This information includes the validity of the token, the information encoded in the token itself, as well as some usage information, i.e. if the token has restrictions regarding the number of usages, information how often it was already used are included. Obtaining the information encoded in the JWT mytoken from the tokeninfo endpoint can be useful when an application only holds a short token instead of a JWT and therefore cannot decode the token itself. This capability is a very basic capability and should generally be included in mytokens.

tokeninfo_history This capability allows querying the mytoken tokeninfo endpoint for the event history of the mytoken. The event history includes entries such as token creation or when the token was used to obtain an access token.

tokeninfo_tree This capability allows querying the mytoken tokeninfo endpoint for a list of mytokens that were created from the presented mytoken. More precisely it returns a tree of subtokens.

list_mytokens This capability allows querying the mytoken tokeninfo endpoint for a list of all mytokens of the user. This capability should only be used for a token if there are good reasons. However, we want to point out that the list of mytokens returned for the list_mytokens and also for the tokeninfo_tree actions does not contain any actual mytoken, only meta-information about the tokens, e.g. the name of a token. Rather, it is the case that it is impossible for the mytoken service to return the actual mytokens, because it does not store them (see section 4.1).

3.2.3. Restrictions

In section 2.2 we motivated the property of refresh tokens being bound to a specific client so that they cannot be passed around, reducing the risk that a stolen refresh token could be used by an attacker. From the requirements it becomes clear that the mytoken service should provide a refresh token like token that can be used to obtain access tokens, but contrary to refresh tokens can be passed around. We already mentioned that we called these tokens mytokens in our service. However, if mytokens are just "refresh tokens

that can be passed around", we work around that security property of refresh tokens and render it useless. Therefore, it is fundamentally important that mytokens can be restricted properly to meet security requirements on the one hand, but also usability requirements on the other hand. We decided to give users the possibility to restrict their mytokens in different ways, combining all of them in what we call *restrictions*.

We stated that the long-running-jobs problem could be easily solved by letting clients request access tokens with a certain lifetime. We like that approach and decided to offer that possibility with mytokens. So one dimension in which users can freely restrict mytokens is time. But mytokens can also be restricted with regards to usage location and number of usages, all under the full control of the user.

As stated in subsection 3.2.1 each mytoken has a restrictions property. This property describes the restrictions of that mytoken. It is a JSON array containing none, one, or multiple *restriction clauses*. A restriction clause is a JSON object that can have any number and combination of the following restriction properties:

nbf The mytoken cannot be used before this time. nbf and exp define the time-span within which the token can be used.

exp The mytoken cannot be used after this time. nbf and exp define the time-span within which the token can be used.

scope Access tokens obtained through this mytoken can only have these scopes.

audience Access tokens obtained through this mytoken can only be used at these audiences.

ip This mytoken can only be used from these ip addresses. It is possible to use single ip addresses or subnet addresses. When requesting a mytoken the special value this can be used and will be replaced automatically with the requester's ip address.

geoip_allow Mytoken uses an ip geo-location database to locate the country from which the request was sent and only allow it if the country is included in this allow list.

geoip_disallow Mytoken uses an ip geo-location database to locate the country from which the request was sent and reject it if the country is included in this disallow list.

usages_AT Restricts how often this mytoken can be used to obtain an access token.

usages_other Restricts how often this mytoken can be used for actions other than obtaining an access token.

As stated, a restriction clause can include any number and combination of these properties. A restriction clause will only *match* (i.e. the operation is allowed) if all of the properties inside the clause match. Logically speaking, the elements inside a restriction clause are ANDed. On the other side all restriction clauses of a mytoken are ORed, i.e. if one restriction clause matches, the operation is allowed. As stated earlier, the exp and nbf

attributes of a mytoken are computed from its restrictions. For nbf the earliest nbf value of all restriction clauses is used, for exp the latest exp value, respectively. If there is at least one restriction clause that does not include a nbf value, the mytoken must be usable as soon as issued, so nbf for the mytoken is set to iat. Similar, a mytoken does not expire, if it has at least one restriction clause without an exp value.

```
{
    "exp": 1620216000,
    "geoip_allow": ["de"],
    "scope": "openid profile",
    "usages_AT": 1
}
```

Listing 3.1: Example restriction clause

A possible restriction clause can be seen in Listing 3.1. With this clause the mytoken can only be used a single time to request an access token with the openid profile scope, only before 2021-05-05 14:00, and only from locations within Germany. Please note that in this example the number of access token requests is limited (only a single request), but the number of other requests is not limited, since the usages_other property is not used. To disallow other usages one has to include

```
"usages_other": 0,
```

in the restriction clause.

By combing multiple restriction clauses, users can create complex, but flexible restrictions. The restriction displayed in Listing 3.2 has three restriction clauses and the associated mytoken can be used if any of these clauses matches. Therefore, it can be used in any of these three cases:

a) From 2021-05-30 14:00 until 2021-05-31 14:00, from the ip 142.42.42.42 and the subnet 142.142.142.0/24, to obtain a single access token with the submit-job scope and the https://hpc.example.com audience.

b) From 2021-05-30 14:00 until 2021-05-31 14:00, from the subnet 142.142.142.0/24, to obtain a single access token with the storage.read scope and the https://storage.example.com audience.

c) From 2021-06-02 14:00 until 2021-06-05 14:00, from the subnet 142.142.142.0/24, to obtain multiple access tokens with the storage.write scope and the https://storage.example.com audience.

If a user knows that a compute job will be started within the first day and that it will run for three to six days, a mytoken with that restriction could easily be used when starting the job. Restriction clause a) allows the user to submit the compute job; the necessary access token can be obtained from the user's machine (142.42.42.42) or from within the

job submission system (142.142.142.0/24). Then, when the job starts, it can obtain an access token to download some data; this is allowed by restriction b). Restriction c) allows to write back the results between three and six days later.

Our flexible restriction approach gives users full control over their mytokens. They can adjust the restrictions exactly to their needs and create complex, tailored solutions, like described in the example. This allows them to restrict the mytoken as much as possible, thereby reducing risks of token abuse, while still allowing operations relevant for the legitimate use.

On the other hand, less advanced users do not have to use the full potential of the restrictions. They still can use one basic restriction clause that can be easily constructed, but already gives substantial security benefits.

3.3. Endpoints and Operations

In this section we describe the endpoints of the designed mytoken service.

3.3.1. Configuration Endpoint

In OIDC each OP has a configuration endpoint. The URL of this endpoint is constructed by appending /.well-known/openid-configuration to the issuer url. This endpoint returns metadata for that OP, e.g. which scope values are supported, but also the URLs of all the other endpoints; therefore, it is useful for discovery.

In a similar fashion, we designed a configuration endpoint for the mytoken service. The URL of this endpoint is constructed by appending /.well-known/mytoken-configuration to the instance's issuer url. The mytoken configuration endpoint returns similar information as the OIDC configuration endpoint, however, it is adapted for the mytoken service. Information returned from this endpoint includes URLs for all the other endpoints as well as information about the supported features, OPs, and operations that can be done. Generally, the configuration endpoint helps clients to discover the other endpoints and available features on the mytoken instance. An example of a mytoken configuration endpoint response can be seen in Listing A.2.

3.3.2. Mytoken Endpoint

Propapbly the two most important endpoints of mytoken are the *mytoken endpoint* and the *access token endpoint* (subsection 3.3.3). The access token endpoint (next section) is properly the most used endpoint. The mytoken endpoint which we present now, on the other hand is the most versatile endpoint, because it can be used in many different ways.

As the name indicates the mytoken endpoint is used to obtain a mytoken token from the mytoken server. This is true for all (successful) requests to the mytoken endpoint. However, as stated, there is some flexibility. This flexibility comes for the request as well as the response.

The final successful response always contains a mytoken. However, in subsection 3.2.1 we introduced the different representations of a mytoken, which all can be requested

from the mytoken endpoint. Hence, the service will return the token either as a JWT (the default), as a short token, or as a transfer code.

On the request side there is more flexibility, because there are different ways to obtain a mytoken, which all require some sort of authorization. However, there are multiple ways for this authorization. In accordance with OIDC we called this "ways of authorization" grant types. In the following we describe these grant types and how they are used:

OIDC Flow The basic grant type is using an OIDC flow. Each user has to do this at least once, because it is the only way[2] for the mytoken service to obtain the required refresh token from the OP. A request with this grant type does not immediately receive a mytoken, because the OIDC flow has to be done first. Generally, the mytoken service first returns a URL that can be used to start the OIDC flow.

When the application that started this flow is a native application, i.e. not a web service, it also receives a polling code. Mytoken implements a flow that is similar to the OIDC device flow. This means that the returned polling code can be used by the native application to poll the mytoken service for the mytoken token and eventually receive it. For services on the web this is not required and they will receive the mytoken directly after the last redirect of the authorization code flow (subsection 2.2.3.1).

The OIDC flow grant type only supports the authorization code flow, because we believe that this is enough. Through the use of polling codes we implemented a way that behaves similar to the device flow for native applications to obtain mytokens also for the authorization code flow. Therefore, the device flow which is supported by less OPs is not needed. However, we designed it in way that it can be easily extended with other OIDC flows, should there be demand.

Polling Code Native applications can obtain a polling code when initiating the authorization code flow. This first request does not return the mytoken, but only the URL to start the OIDC flow and a polling code for the application. Using this code the application can poll the mytoken endpoint. Similar to the device flow, the mytoken service returns a message indicating that authorization is still pending while this is the case. When mytoken receives a refresh token from the OP after the authorization code flow was completed and the mytoken token was created, the next polling request will receive the mytoken.

Transfer Code Transfer codes returned from the mytoken endpoint or transfer endpoint (subsection 3.3.4) can be exchanged into a mytoken. A request using the transfer code grant type is very simple, because it simply exchanges the transfer code for the associated stored mytoken.

Mytoken With the mytoken grant type, a mytoken token with the `create_mytoken` capability can be used to create a new mytoken. This is useful, because it allows users to easily create multiple mytokens, so that a mytoken can be created and passed to another

[2] In a possible future extension mytoken could also use the token exchange flow to obtain a refresh token from a presented access token. However, we decided to not implement this at the moment, because many OPs do not support token exchange and we want mytoken to be as general as possible.

application, e.g. to submit a compute job, but it does not require the user to do an OIDC flow again.

Mytokens created from another mytoken cannot be more powerful than the mytoken they were created from. This is an important security property, so that token privileges cannot be escalated. On the contrary, it is possible to restrict a mytoken further. This allows users to have a rather powerful "master" token from which they can create other more restricted mytokens that can then be passed to other applications or machines. Finally, when a mytoken M is created, it is possible to specify a set of capabilities as subtoken_capabilities. Any mytoken created from M can only have capabilities that are included in this set.

Consider the following mytoken M_p:

```
{
  "capabilities": ["create_mytoken"],
  "subtoken_capabilities": ["AT", "tokeninfo_introspect"],
  "restrictions": [
    {
      "usages_AT": 5
    }
  ],
}
```

This token M_p can create other mytokens with the AT and tokeninfo_introspect capabilities. It can create tokens with these capabilities even tough the token M_p itself does not have them, because they are listed as subtoken_capabilities. Note also that M_p has a restriction that restricts the number of access tokens that can be obtained to 5. However, because M_p does not have the AT capability it is not possible at all to obtain an access token with M_p itself. This restriction is still reasonable, because the mytokens created from M_p will inherit it. Using this pattern, a user can create a token M_p that can be used to create mytokens which can only be used five times for obtaining access tokens. If multiple mytokens are created from M_p, each one can obtain up to five access tokens.

Of course it is also possible to specify additional restrictions for the newly created mytoken. The following restriction would be accepted by the mytoken service for a mytoken created from M_p:

```
{
    "usages_AT": 1,
    "scope": "openid profile email"
}
```

However, a restriction that contains only the part with the scope and does not include any restriction for usages_AT would not be accepted by the mytoken service, because the requested restriction is not tighter than the restriction of M_p. The restriction is not tighter, because M_p is restricted with usages_AT while this is not the case for the requested restriction. In this simple example it would be possible to combine the restrictions of M_p with the requested restrictions and merge them together, but generally this is not

possible, especially when there are multiple restriction clauses. Therefore, mytoken does no automatic merging, but will either use the original restriction or fail with an error[3].

3.3.3. Access Token Endpoint

The access token endpoint returns an OIDC access token when a valid mytoken is presented. The mytoken service checks the mytoken and uses the associated refresh token to obtain a new access token which is then returned to the client. The "refreshing" of access tokens via a mytoken is quite similar to refreshing via a refresh token.

It is possible to request access tokens with specific scopes and audiences from the access token endpoint. However, naturally the requested scopes and audiences must be permitted by the used mytoken.

3.3.4. Transfer Endpoint

We already mentioned, that one can obtain a transfer code from the mytoken endpoint instead of the created mytoken. This can be useful if one wants to create a new mytoken on one machine, but needs to use it on another; then the user can directly obtain a transfer code for that token and use that code on the other machine to obtain the actual mytoken.

There are also cases where users may want to transfer an already existing mytoken. This can be done using the transfer endpoint. A request to it must contain the token that should be transferred. The mytoken service then creates a transfer code for the token, stores the token in an encrypted way, and returns the transfer code. To obtain the token (on the other machine) the transfer code can be exchanged at the mytoken endpoint.

3.3.5. Revocation Endpoint

Long-lived credentials such as refresh tokens and mytokens must be revocable. For this, the mytoken service provides a revocation endpoint where any token issued by the service can be revoked. Mytokens can be sent as JWTs or as short tokens - since short tokens can be used just like JWT mytokens. After sending a token to the revocation endpoint, it will be revoked and cannot be used anymore. When revoking a mytoken that was used to create other mytokens (subtokens), it is possible to only revoke the send token or to additionally revoke all subtokens (recursively). Even though transfer codes are not long-lived and have a very limited lifetime, these are also revocable.

3.3.6. Tokeninfo Endpoint

The tokeninfo endpoint can be used to obtain various information about a mytoken. The endpoint supports different actions, each requiring its own capability. We already described these actions when we introduced the available capabilities in subsection 3.2.2.

[3]The default is to use the original restriction, but one can request that an error should be returned instead.

```
[
    {
        "nbf": 1622376000,
        "exp": 1622462400,
        "scope": "submit-job",
        "audience": [
            "https://hpc.example.com"
        ],
        "ip": [
            "142.42.42.42",
            "142.142.142.0/24"
        ],
        "usages_AT": 1,
        "usages_other": 0
    },
    {
        "nbf": 1622376000,
        "exp": 1622462400,
        "scope": "storage.read",
        "audience": [
            "https://storage.example.com"
        ],
        "ip": [
            "142.142.142.0/24"
        ],
        "usages_AT": 1,
        "usages_other": 0
    },
    {
        "nbf": 1622635200,
        "exp": 1622894400,
        "scope": "storage-write",
        "audience": [
            "https://storage.example.com"
        ],
        "ip": [
            "142.142.142.0/24"
        ],
        "usages_other": 0
    }
]
```

Listing 3.2: A restriction with multiple restriction clauses

4. Implementation

In this chapter we describe some important and interesting aspects of the implementation of the mytoken service. Please note that we only cover some aspects of the implementation in this chapter, but everything described in the previous chapter was implemented accordingly.

The full implementation of the mytoken server can be found at https://github.com/oidc-mytoken/server and the command line client at https://github.com/oidc-mytoken/client.

We implemented both the server as well as the command line client using the Go programming language [25]. Go has great concurrency- and cross-platform-support. This enables us to easily provide our software for multiple platforms. This is especially useful for the client, since it needs to run on large variety of operating systems. The server builds on the fiber web framework[1] which offers good tooling and great performance. A performance benchmark has been added in Figure A.1.

4.1. Encryption of OpenID Connect Tokens

One important implementation aspect is the encryption of OpenID Connect (OIDC) tokens, especially refresh tokens. The central design decision regarding token encryption was to store refresh tokens in a way that they can only be accessed when needed and when appropriately authorized. This means that the refresh tokens need to be stored in an encrypted way in the database and that they can only be decrypted on the user's demand. This adds an additional layer of security. Even if an attacker gets full access to the database, the stored tokens cannot be abused, because they cannot be decrypted. In particular even administrators are not able to decrypt the stored refresh tokens. If an administrator could do this, they could use the refresh token together with the client credentials and impersonate any user. Therefore, the encryption of refresh tokens is a security measure against outside and insider attacks. However, mytoken still must be able to decrypt the refresh token, when a user requests a new access token to fulfill the request.

Implementing this feature had big impacts on the whole implementation and some design decisions. Therefore, we detail the used concepts and implications in the following.

From the previous description of this feature we can formulate two requirements:

- Decryption must not be possible with full access to the database.
- Decryption must be possible on user's request.

[1] https://gofiber.io/

From these two requirements it becomes clear that only the user's request allows decryption, i.e. the user must provide the necessary information for decryption. Also, this information must not be stored in the database.

Since the flow to obtain access tokens from a mytoken server is very simple, the only suitable piece of information is the mytoken itself. The straightforward approach would be to use the mytoken directly as the decryption key[2]. This means that the refresh token R is encrypted with the mytoken M when it is obtained and the mytoken is created. We denote the encryption of a token T with an encryption key K as $E(T, K)$. The encrypted refresh token $R_c := E(R, M)$ is stored in the database (linked to the mytoken M), but the actual mytoken M is not stored (only its metadata, such as the token id). On access token requests the mytoken server uses the mytoken from the request to decrypt the encrypted refresh token and uses that to obtain an access token.

While this approach is simple and clean, it is not suitable in our case: When a mytoken M_p is created through an OIDC flow the server receives a refresh token R. This refresh token is linked to the created mytoken M_p and is required for obtaining access tokens. If a new mytoken M_n is created from the mytoken M_p, M_n still needs a linked refresh token to be able to obtain access tokens later. Because no OIDC flow is performed, the refresh token cannot be obtained from the OpenID Provider (OP)[3]. Therefore, the new mytoken M_n must use the same refresh token R as its parent M_p. Linking the refresh token R to M_n can be implemented in two ways:

Link by reference This means that M_n is linked to $R_{c,p} := E(R, M_p)$ - an encryption of R that was created with M_p. Therefore, M_n cannot be used to obtain access tokens when it is linked to $R_{c,p}$ because $R_{c,p}$ cannot be decrypted with M_n. Consequently, this approach is infeasible.

Link by value This means that $R_{c,p}$ is temporarily decrypted using M_p in order to create M_n. $R_{c,p}$ will not be changed and remains stored in the database linked to M_p. The decrypted refresh token R is then also encrypted using M_n and stored linked to M_n. Consequently, there are two ciphertexts of R stored in the database: $R_{c,p} := E(R, M_p)$ and $R_{c,n} := E(R, M_n)$. This allows that both tokens M_p and M_n can be used to obtain access tokens; both can decrypt their own ciphertext of the refresh token R.

While this works perfectly fine for everything presented so far, it still is not perfect. This implementation has a problem when the refresh token changes. While this may not yet be commonly observed, the specification allows it. Furthermore, OPs are encouraged by security researches to implement token rotation, i.e. each refresh token can only be used once and the response contains a new refresh token. If an OP implements token

[2] Actually it is not directly used as the decryption key. It is more used as a decryption "password" from which the actual key is derived. However, we omit this detail here and in the following and refer to it directly as the encryption / decryption key for better readability.

[3] With token exchange the mytoken server could obtain a new refresh token for the new mytoken, so that each mytoken has a dedicated refresh token linked. This would solve the in the following introduced update-problem, but would introduce a similar problem when doing a recursive mytoken revocation. Furthermore, token exchange is not widely supported and we do not wan to require it for this basic feature.

rotation or a refresh token changes for other reasons, the above explained implementation of refresh token encryption is not sufficient. Since M_p and M_n both use the same refresh token R, the ciphertexts must be updated for both when R changes. However, while it is possible to discover that there are other copies of the same refresh token stored (to do so, one would additionally store a hash value of the refresh token), there is no way to update the other copies correctly[4]. (By correctly we mean that they can be decrypted with the mytoken they are linked to.)

To also support this case where an encrypted refresh token changes, a different implementation strategy is required. We therefore adapted the strategy, described above, slightly by combining it with the link-by-reference approach:

Using dedicated encryption keys Only one ciphertext R_c of a refresh token R is stored in the database. This copy is linked to all mytokens that use it (M_p and M_n in the above example). The refresh token R is encrypted with a dedicated encryption key K created by the mytoken server, therefore $R_c := E(R, K)$. The encryption key K is then encrypted with the mytoken (for each linked mytoken). This means that when M_p is created, the refresh token R is obtained from the OP, it is encrypted with a freshly created encryption key K, the encryption key K is encrypted with the mytoken M_p, and the ciphertexts $R_c := E(R, K)$ and $K_{c,p} := E(K, M_p)$ are stored in the database. When the mytoken M_n is created, the encrypted refresh token R_c is not needed. Only $K_{c,p}$ is decrypted using M_p, which gives K that is then encrypted with M_n. At the end $K_{c,p}$ remains stored unchanged and $K_{c,n}$ is stored linked to M_n.

This allows that both tokens M_p and M_n can decrypt their copy of the encryption key and then decrypt the refresh token when needed. Also, if the refresh token changes, it can be easily updated by any of the linked mytokens. A mytoken M just updates the refresh token cipher $R_c := E(R, K)$ using the encryption key K which is shared by all mytokens linked to the same refresh token. This approach is visualized in Figure 4.1.

When mytoken creates a short token (subsection 3.2.1) it uses a JSON Web Token (JWT) mytoken internally and maps it to a random string - the short mytoken. This allows mytoken to handle short and long mytokens in just the same way. When a short mytoken is used the mytoken server just has to get the linked JWT mytoken from the database. Therefore, mytoken still uses the JWT mytoken as an encryption key. However, because of this fact, the JWT cannot be stored in plaintext in the database. Therefore, the short token S is used to encrypt the long mytoken as $M_c := E(M, S)$. This is visualized in Figure 4.2a.

When a native client application initiates the authorization code flow to obtain a mytoken, it cannot obtain the mytoken directly. For web applications this is different; these receive the mytoken at the end of an http redirect. Since we cannot redirect to a native application, these receive the mytoken through polling. For this purpose the application receives a polling code from the mytoken server when it starts the flow. As there will be some time gap between the creation of the mytoken and the time when the application

[4] Assuming symmetric encryption. Asymmetric cryptography would be a solution for this particular problem, but is generally not suitable.

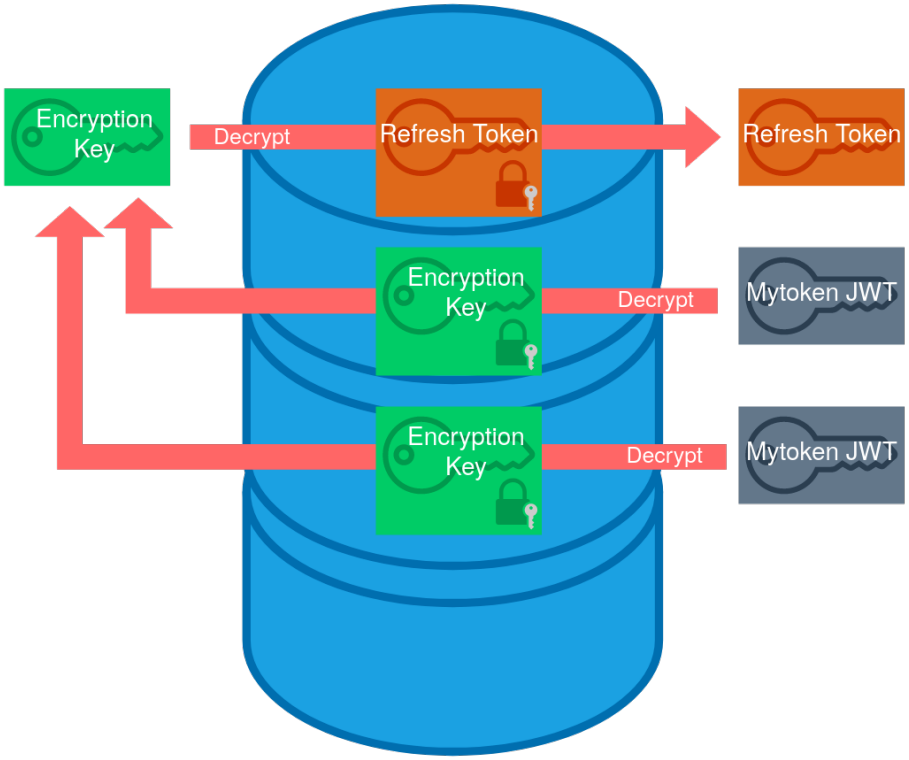


Figure 4.1.: Storing an encrypted refresh token with multiple linked mytokens

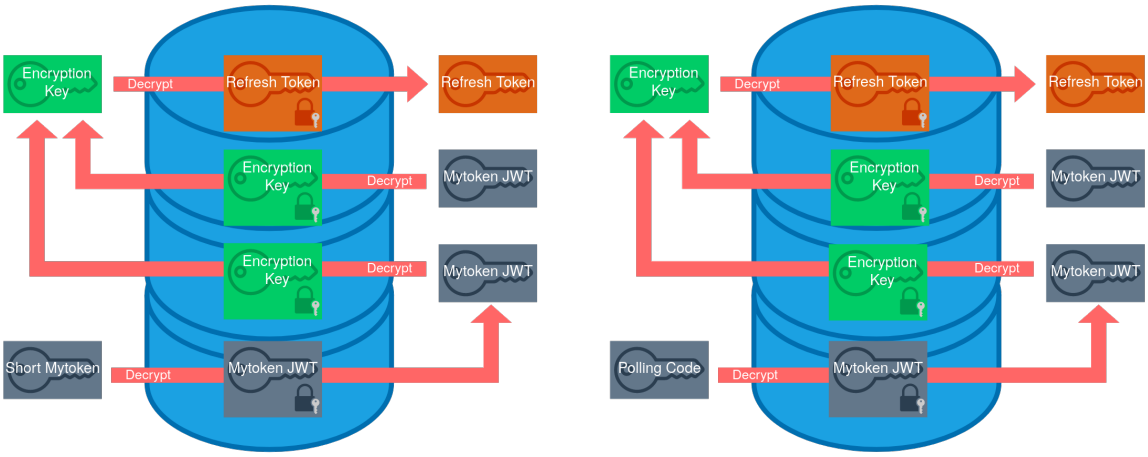
successfully polls, the mytoken must be stored. Since the mytoken is used as an encryption key it cannot be stored in plaintext. This is analog to the previous case with a short token. Using the same straightforward approach, the mytoken server uses the polling code P to encrypt the mytoken M, so that $M_c := E(M, P)$ can be stored in the database. This is also visualized in Figure 4.2b.

The last case is only secure[5], if the polling code is not stored in the database, in which case it could easily be used to decrypt the linked mytoken and allow access to the linked refresh token.

In a similar fashion, we also must ensure that the data stored in the database is not enough to re-create the JWT mytoken. If all data encoded in the JWT are stored in the database, an administrator with access to the private key used for signing the mytokens could easily create any mytoken from the stored data.

We solve both problems using the same technique. The root of the problem is that mytoken needs a reference from one piece of information to another (i.e. the mytoken id must be linked to the other token properties), but at the same time the information cannot be stored as it is, because that would allow decrypting sensitive information. We can elegantly solve these problems by using a cryptographic hash function. Instead of storing the original data (e.g. the token id) mytoken only stores a hash value of it. This

[5]By secure we mean that it fulfills our requirement that no one with full database access can decrypt refresh tokens.



(a) Storing an encrypted refresh token with a linked JWT mytoken and a linked short mytoken

(b) Storing an encrypted refresh token with a linked JWT mytoken and a linked polling code

Figure 4.2.: Storing encrypted refresh tokens with mytokens, short tokens, and polling codes

way it is impossible[6] to recreate the original value from the hash value, i.e. to obtain the decryption key. But when the user presents the original data (e.g. the mytoken JWT with the encoded id) that can be used to obtain the hash value and match it to the stored data.

Mytoken uses these technique multiple times; as stated it is used for polling codes as well as mytoken ids. When it comes to polling codes, mytoken also stores other information that must be linked to the polling code (e.g. the OIDC state). The same implications apply here, so that mytoken only stores a hash value of the state value. However, the state and polling code must be linked in a way that the polling code can be created from the state. This is necessary, because the mytoken is created after the code exchange step of the authorization code flow, at this time the mytoken server does not have access to the polling code to encrypt the mytoken. However, if the polling code can be created from the state (which mytoken has access to on code exchange) this problem can be solved.

Since mytoken already stores the hash of the state in the database, this value cannot be used as the polling code. So instead another way to create the polling code from the state value must be used. There are multiple ways to do so, we decided to create the polling code as a Message Authentication Code (MAC) from the state, in particular we use a Hash-based MAC (HMAC). Figure 4.3 visualizes the storage of the hashed values in the database and how all of these values are linked.

The encryption of refresh tokens in a way that not even administrators can decrypt them is a nice and desirable feature. But, it also should have become clear, that this seemingly

[6]To be more precise: For a cryptographic hash function H and a probabilistic polynomial time adversary $\mathcal{A}$, the probability $\Pr[X' \leftarrow \mathcal{A}(H(X), 1^k) : H(X) = H(X')]$ that $\mathcal{A}$ breaks the pre-image resistance of H is negligible.

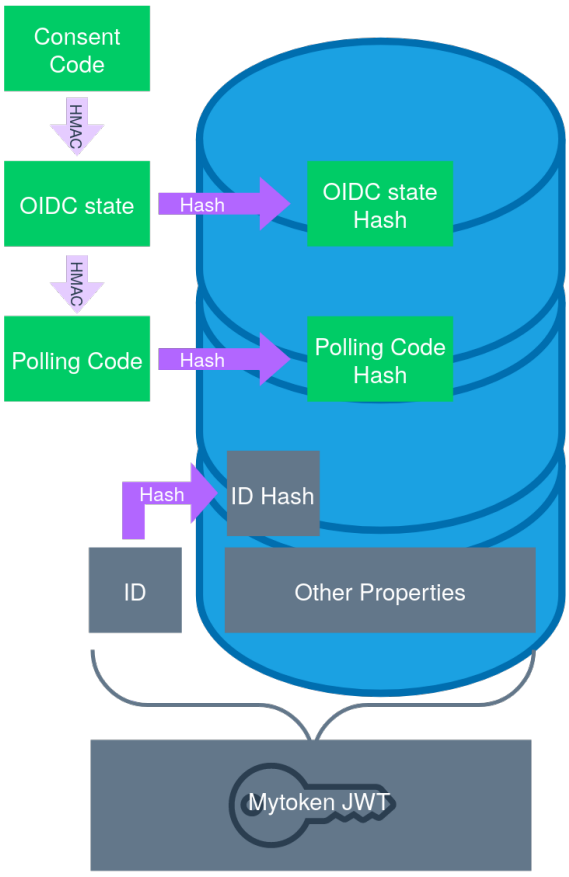


Figure 4.3.: Storage of hashed secrets in the database

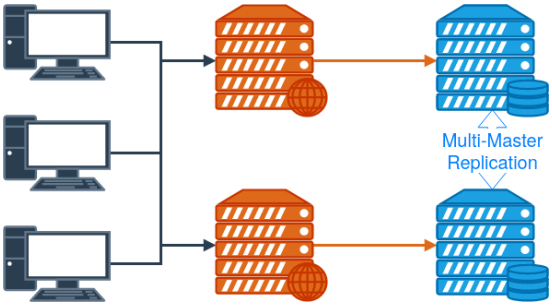
easy to implement feature is more complex than it seems to be and the implementation impacts other features such as short tokens, polling codes, and token revocation.

4.2. High Availability

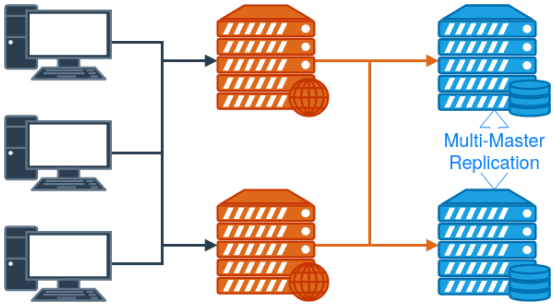
We designed and implemented the mytoken service in a way that it can support high availability. In the following we refer to the mytoken service as frontend and the database as backend. It is possible to deploy multiple frontends with the same configuration on different machines. If they connect to the same database (and use the same configuration, e.g. issuer URL), users can connect to any frontend instance. Every frontend instance is perceived as the same mytoken instance from a user perspective. Each frontend instance may fail at any time without an effect on the other frontends. No state is held in the frontend instances, only in the database. As long as there is at least one frontend instance available, the service as a whole is operational.

It is also possible to deploy multiple backends. However, the frontend does not handle synchronization, this must be done by the backends. Mytoken expects that it can read and write to any backend. One can deploy mytoken with multiple frontends each connecting to one backend node (Figure 4.4a) or to multiple backends (Figure 4.4b). If a frontend

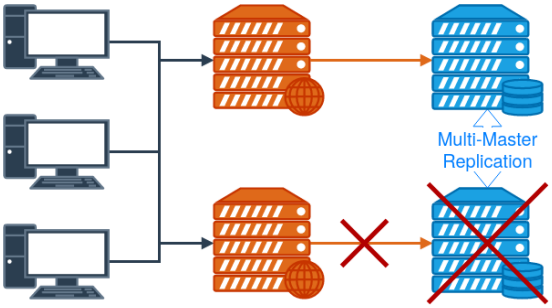
has multiple backend nodes configured it will cycle through them when accessing the database. If a backend node fails (Figure 4.4d), the frontend does no longer consider it in the round-robin until it comes back. The frontend is able to handle failed backend nodes and will try to reconnect to a failed node until it comes back. If a frontend is only connected to one backend and that backend fails (Figure 4.4c), the frontend can no longer operate as expected. A load balancer should then exclude this frontend node from load balancing; however, additional logic is required, to discover this case.



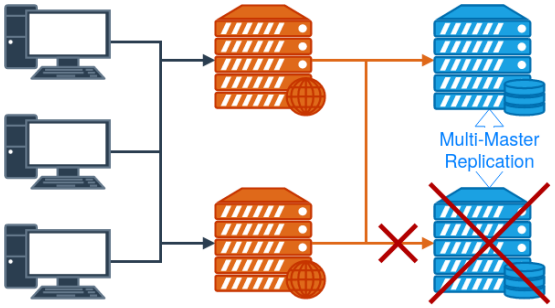
(a) High availability setup with one backend per frontend



(b) High availability setup with multiple backends per frontend



(c) Backend failure in a high availability setup with one backend per frontend



(d) Backend failure in a high availability setup with multiple backends per frontend

Figure 4.4.: High availability setups with dedicated backends for each frontend and frontends connect to all backends

However, we recommend running mytoken not with dedicated backends per frontend, but connecting each frontend to all backend nodes. This way each backend and frontend might fail independently and the service is still operational if at least one frontend and one backend are up.

As already stated the mytoken frontends expect that they can write to any backend node and that the database handles replication. Such a multi-master replication is provided by the Galera Cluster[38] for MySQL[7] and MariaDB[8]. We verified that mytoken works with the Galera Cluster and MariaDB. We found both, independent frontends as well as independent backends, working as expected.

[7] https://www.mysql.com/
[8] https://mariadb.com/

4.3. Command Line Client

We implemented a command line client which is also called mytoken. The client can be used to easily obtain mytokens and access tokens from a mytoken server. The mytoken client can also manage the obtained mytokens.

In the following we shortly cover the most important features of the mytoken client by describing the three most used commands.

4.3.1. Storing Mytokens

Using the MT store command, the mytoken client obtains a mytoken and stores it in an encrypted way. Mytokens obtained with the MT store command are managed by the client and not displayed to the user. Users can configure the encryption of mytokens: They can either use a GPG-key to encrypt the token using the user's gpg-agent[9], or alternatively, a password may be used to encrypt the token with AES256.

A mytoken can be obtained from an existing token or by performing an OIDC authorization flow. If a mytoken is used to generate a new one, it can be passed via the command line (not recommended), stdin, a file, or an environment variable; it is also possible to use a mytoken managed by the client. The examples in subsection 4.3.2 and subsection 4.3.3 use a passed and already stored mytoken, respectively. The following example command line starts an OIDC flow to obtain a mytoken. The new mytoken is encrypted and stored, associated with the name "master".

```
mytoken MT store --oidc --usages-at=5 --ip=this --exp=+1d master
```

The mytoken stored with this command can then be easily used with the mytoken client. The example for obtaining access tokens will cover this.

It can be seen from the above command line that restrictions for the mytoken can be passed via command line options. In this example the number of times that an access token can be obtained is restricted. Furthermore, it can only be used from the same ip address where it was requested, and it expires after one day. The command line options that take a time value accept the following date specifications:

Numeric UNIX Timestamp This is the way in which mytoken handles the date values on the Application Programming Interface (API). However, for users this is mostly not suitable.

Absolute Time Passing the time as a formatted string (e.g. "2021-05-05 14:00") is much more user-friendly than passing it as a UNIX timestamp. This way can be used if the token should expire at (or not be usable before) a given point in time.

Relative Time A relative time value must be prefixed with "+", e.g. "+1d6h30m". This is probably the most useful method, since it allows users to create tokens that are valid *for* a given time, rather than valid *until* a given time. It is also the most useful method for usage in scripts.

[9] https://gnupg.org/

4.3.2. Obtaining Mytokens

The MT command is used to obtain a mytoken. Here, by "obtaining" we denote obtaining a mytoken from the mytoken server and returning it to the user. The obtained mytoken will not be managed by the mytoken client.

The following command line obtains a mytoken from an existing mytoken that was previously stored in a file (by the user).

```
mytoken MT --MT-file=/tmp/mt --restrictions=/tmp/restrictions
↪  --token-type=short
```

In the above command the mytoken is requested as a short mytoken using the --token-type option. In the example in the previous section, specific command line arguments, such as --exp or --usages-at have been used to specify restrictions for the mytoken. In the above example, the --restrictions option is used. This requires users to specify their restrictions as JSON which allows more complexity. The JSON restriction can then be either passed directly to the option, or via a file as it is the case in this example.

By default, the returned mytoken is simply displayed to the user, but it is also possible to write it to a file. But it then is still not encrypted; it is the user's responsibility to handle mytokens securely, if they are only returned and not stored / managed by the client application.

4.3.3. Obtaining Access Tokens

Obtaining access tokens from the command line is obviously an important feature of the mytoken client. If a user has configured a default OP and a default mytoken for that OP, obtaining an access token using these parameters is as easy as:

```
mytoken AT
```

A more complex example is:

```
mytoken AT -i wlcg -t master -s openid -s profile --aud="https://example.com"
```

Both of the just presented commands use a mytoken stored and managed by the client. In the first example the default token for the default OP is used. In the second example the OP (wlcg) and mytoken (master) are explicitly passed, as well as the scopes and audiences for the access token that should be obtained.

The obtained access token is printed out to the terminal by default, but it can also be saved in an environment variable or file.

5. Evaluation

In this chapter we evaluate the developed mytoken service and command line client against the requirements and other related software.

5.1. Fulfillment of Requirements

In this section we revisit the requirements from section 3.1 and evaluate if and to which extend mytoken fulfills these.

The two main requirements were:

Command line usage The mytoken client interacts with the mytoken server and enables users to easily obtain mytokens as well as OpenID Connect (OIDC) access tokens on the command line. Furthermore, the implementation of a device flow like flow using polling codes enables usage of the mytoken client on any (command-line capable) device, since the authorization can be done on another device. A web-interface for the mytoken service was out of scope of the thesis. Mytoken has a web-interface which offers similar features as the command line client. This web interface is not part of the thesis and was developed at the Steinbuch Centre for Computing (SCC).

Machine independence Mytokens can be used on any machine to obtain access tokens and can be easily transferred from one machine to another.

The other requirements were:

Security When developing mytoken, security was prime goal. Mytoken uses approved algorithms like AES256, SHA512, and ES512 from Go's standard cryptographic library. We also implemented the encryption of the stored refresh tokens in a way that they cannot be decrypted without access to the linked mytoken (section 4.1).

Provider support Mytoken was developed to support a wide range of OpenID Providers (OPs). We decided to build on core OIDC features like the authorization code flow and refresh flow and not rely on more advanced but not widely supported features like token exchange. However, the restriction of access tokens with an audience is only supported for OPs that implement this feature. We tested the mytoken service with multiple OPs, among these INDIGO IAM[1], EGI Check-in[2], KIT[3], HIFIS[4], and Google[5].

[1] https://wlcg.cloud.cnaf.infn.it/
[2] https://aai-dev.egi.eu/oidc/
[3] https://oidc.scc.kit.edu/auth/realms/kit
[4] https://login.helmholtz.de/oauth2
[5] https://accounts.google.com/

Multiple providers per instance A single instance of mytoken can support multiple OPs. The demonstration instance of mytoken[6] currently supports EGI Check-in and an INDIGO IAM instance for WLCG.

Configurability Both users and administrators can configure there side:

Deployment Administrators can configure the deployed instance according to their needs. In particular administrators are free to enable and disable certain features, like short tokens or transfer codes. They can also set security and privacy options according to their policy; this includes for example the length of codes and short tokens or the signature algorithm used for token signing.

User When a user creates a mytoken they have a lot of control over it: The user can freely choose capabilities and the restrictions of the mytoken. The restrictions are a powerful tool to balance the power of individual mytokens. In addition users can configure their mytoken client with sensible defaults according to their needs.

User-friendliness Mytoken offers an easy installation and an extensive documentation for getting started. This requirement especially focuses on the following two additional points:

Clear interface The command line interface is as clean as possible while supporting the relevant command line options. Through the use of sensible defaults a lot of options can be omitted, so that the commands can become very slim.

Sensible defaults We use sensible defaults also on the server side, where this is applicable (e.g. using the AT capability as default for mytokens). However, this part of the requirement applies much more to the client application: The mytoken client has sensible defaults for various options (e.g. capabilities), which allows omitting these options most of the time. While the defaults are suitable for most users, users can freely change them if they have other needs.

Extensibility We designed the mytoken service in a way that it can be easily extended. For example the restrictions are designed such that it is easy to add other dimensions in which mytokens can be restricted. It is also possible to add additional ways to obtain mytokens from the service. Additional possible extensions are discussed in section 6.2.

Performance Mytoken uses the fiber web framework, which offers good performance. The performance evaluation shown in Figure 5.1 indicates a similar performance also for mytoken. Performance will be further discussed in section 5.2)

High Availability As we have shown in section 4.2, multiple instances of mytoken can be used with a Galera database cluster to support high availability.

Permissive open source license Mytoken (server and client) is released under the MIT license, which is one of the most permissive open source licenses.

[6] https://mytoken.data.kit.edu

Self-hostable Mytoken can be hosted by others. The demonstration instance hosted by KIT is only for demonstration purposes. To help communities host their own instance, we provide a setup tool and extensive documentation, that both help with an easy setup as described by the deployment guide[7].

All in all, it can be said that the developed mytoken service fulfills all the requirements identified in section 3.1.

5.2. Comparison with Related Work

In this section we evaluate mytoken with the two related software projects that we introduced in section 2.3: oidc-agent and htgettoken.

Distribution / Availability The Mytoken client benefits from the cross-platform support of the Go language, so that it is available for a wide range of architectures and operating systems. On Linux, debian- and RPM-based distributions benefit from packaged versions which offer the easiest installation. For other systems (including Windows and MacOS) compiled binaries are provided. Debian packages are available at http://repo.data.kit.edu and all client release artifacts can be downloaded from https://github.com/oidc-mytoken/client/releases.

oidc-agent is available as an RPM and debian package. Special focus has been put on the packaging for debian-based distributions so that various versions of debian and ubuntu are supported. There are current efforts to include oidc-agent in the official debian package repository. It is already available on the arch linux user repository and on gentoo linux. A MacOS version is available via homebrew.

htgettoken as a packaged version is currently only available as an rpm package. However, since the htgettoken client is a python script it should also be usable on other systems when the necessary libraries are installed.

Library support The mytoken project includes a Go library. This library is used by the mytoken command line client, but it can also be used by other applications to obtain tokens from a mytoken server.

oidc-agent provides libraries for C, Go, and Python. A number of clients already use one of these libraries to interact with an oidc-agent.

htgettoken does not provide any library for other applications. However, it should be possible to utilize the python source code.

[7] https://mytoken-docs.data.kit.edu/server/intro/

Performance We tested obtaining access tokens with all three applications against the INDIGO IAM instance for WLCG[8]. This is the only OP supported by htgettoken we have access to. However, this instance sometimes responded unusually slow. We therefore eliminated some outliers from the collected data which clearly came from such delayed responses. The distribution of the response times over 83 remaining data points for all of the three applications can be seen in Figure 5.1. The graph also shows the response time distribution for a curl command that obtains an access token directly from the OP using the refresh flow.

We expected oidc-agent to be the fastest implementation (excluding the curl command), due to the performance of the C language and the minimal overhead compared to a pure OIDC refresh flow. When a refresh token is already loaded in the agent (as it was the case with our test) the overhead added by oidc-agent is minimal and the local agent directly contacts the OP. Both other applications first connect to their server part which then connects to the OP.

It can be seen that the oidc-agent response times are indeed the lowest of the three applications and are very similar to the ones achieved with the curl command. However, while the applications can be ranked where oidc-agent is the fastest, then mytoken, and htgettoken last, the differences between the three applications are only small. Considering the large deviations within each application's response times (which are likely caused by network and OP latency), the performance differences between them are not significant.

We therefore infer that the overhead added by the network requests to the mytoken and htgettoken vault servers as well as the overhead of the processing done at their servers (including decryption) is negligible compared to the network request to the OP and its processing.

While there is no clear performance winner between the three applications, even tough oidc-agent is generally a bit faster, the reverse is also true: None of the applications has significant performance flaws.

Server Security mytoken was not built using any existing secret manager and therefore has to implement the handling of secrets such as refresh tokens. We did not implement any cryptographic algorithms ourselves, but relied on existing libraries. Go offers an extensive standard library that also implements cryptographic algorithms. For the encryption of refresh tokens mytoken uses AES256 from the Go standard library. Also refresh tokens are encrypted such that they cannot be decrypted without the linked mytoken, even if full access to the database (as described in section 4.1) is gained.

For oidc-agent the server security is less crucial, because all data is stored only on the user's machine and not on a remote server. This means that no other person has access to the data; granted that access to the user's machine is well controlled. However, oidc-agent still stores its configuration in an encrypted way. For encryption the popular and well maintained libsodium[9] library and the XSALSA20 algorithm is used.

[8] https://wlcg.cloud.cnaf.infn.it/
[9] https://github.com/jedisct1/libsodium

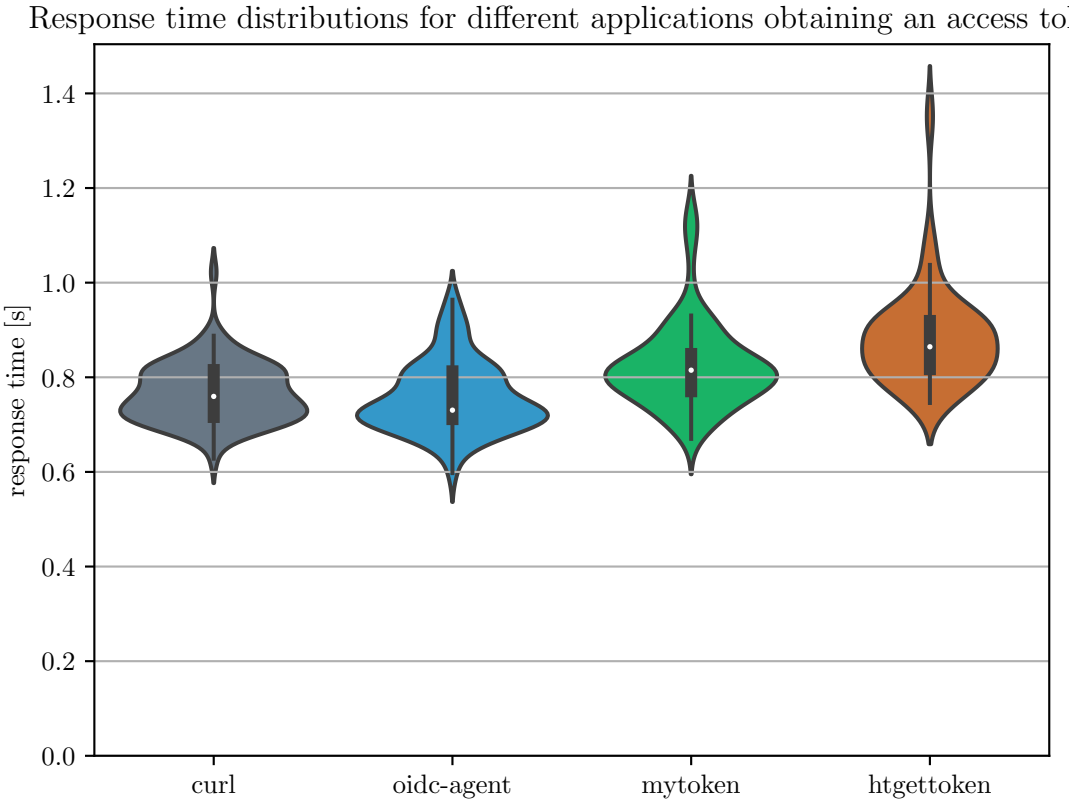


Figure 5.1.: Comparison of the response time distributions of curl, mytoken, oidc-agent, and htgettoken when obtaining an access token for the INDIGO IAM instance for WLCG

htgettoken uses the popular secret management service vault to store refresh tokens and to handle the access to them. Vault is written in Go and uses the same cryptographic libraries as mytoken and also uses AES256 [33].

Client Token-Management The mytoken command line client can be used to manage the obtained mytokens. If mytokens are managed by the client they are stored in an encrypted way on the user's local disk. Two modes for encryption are supported: Using gpg for encryption through the user's gpg-agent or using AES256 with a user provided password. Mytoken additionally supports returning mytokens without storing them in which case the user is responsible to handle them securely. This mode is intended to be used with less-privileged mytokens that eventually expire, so they can be passed to other applications, e.g. to submit a compute job.

For oidc-agent this point is not applicable, since no special token is created that must be handled by the client. The account configuration file, which among other information contains the refresh token, is encrypted as described in Server Security (section 5.2).

The htgettoken client stores the vault token in an unencrypted way. However, its lifetime is limited to 11.5 days due to infrastructure security policies.

Documentation mytoken provides documentation as part of the client's help output, but also has an extensive online documentation[10] that explains the principles such as mytokens and restrictions. It also includes a walk-through for client usage, and provides an example configuration file with explanatory comments.

oidc-agent provides helpful information in the help outputs of the different tools, which all also have a man page, and there is an extensive documentation available[11].

htgettoken provides a help output and a man page. No additional documentation page is available.

Token Restrictions mytoken offers the *restrictions* mechanism for users to restrict their mytokens freely. Mytokens can be restricted in their lifetime (both starting and expiration), as well as scope and audience restrictions, restrictions where they can be used (ip, geo ip), and in the number of usages.

oidc-agent offers to set the scopes and audiences of a refresh token and also the same for the access tokens. Because there is no other special token type involved other restrictions are not used.

htgettoken offers setting the scopes and audiences, as well as the lifetime of a vault token.

Usability for Obtaining Access Tokens This aspect focuses on the usability to obtain access tokens, other features are ignored, if not relevant to obtaining access tokens. With all applications users have to do some additional setup, which will always involve an OIDC authorization flow. This is required by the nature of OIDC. In this aspect all three applications are similarly user-friendly. However, oidc-agent might be a bit more complicated depending on the OPs, if no preregistered client can be used. In the following we focus on the usability of the command to obtain access tokens.

mytoken can be used very easily to obtain access tokens if a mytoken was already obtained. Since the lifetime of a mytoken can be freely chosen by the user and additional mytokens can be created from an existing one, users usually have to do the initial OIDC flow only once. When obtaining an access token special scope and audience values can be easily requested using command line options. To request multiple values the option has to be passed multiple times. An example command is the following:

```
mytoken AT -s openid -s profile -s storage.read
↪  --aud="https://storage.example.com"
```

[10] https://mytoken-docs.data.kit.edu/
[11] https://indigo-dc.gitbook.io/oidc-agent/

If the token was encrypted using a password, the password has to be entered every time the token is used, i.e. also every time when an access token is requested. However, this is not required if the mytoken was encrypted through the gpg-agent or if a mytoken is directly passed.

oidc-agent can be used with oidc-token in a very similar way to mytoken when it comes to passing scope and audience values, as can be seen from the following example:

```
oidc-token wlcg -s openid -s profile -s storage.read
↪  --aud="https://storage.example.com"
```

A user has to provide either the issuer URL (then the default account configuration for that issuer is used) or a short name identifying the used account configuration file. Also, the used account configuration must be loaded in the agent in order to be able to obtain an access token. However, on desktop systems, i.e. systems with a graphical user interface, it is possible to automatically load an account configuration when required and it is not already loaded. Usually, an account has to be loaded only once after the agent was started, for example it must be loaded again after a system reboot; however, this is a fairly small burden. While oidc-agent offers support for agent forwarding through ssh so that a local oidc-agent can be accessed from the remote server, this is only limited and not enough for all use cases. It is also possible to copy an account configuration to another machine and use it there, however, this is also not suitable in all cases, e.g. on a shared system.

The htgettoken command for obtaining an access token with specific scope and audience values is:

```
htgettoken -a vault.example.com --scope="openid profile storage.read"
↪  --audience="https://storage.example.com" -o /dev/stdout
```

The URL of the vault server passed with the -a option, can also be set with an environment variable. To be more precise, the environment variable HTGETTOKENOPTS can be used to set options that should be included with all htgettoken calls. Unlike mytoken and oidc-agent, htgettoken takes multiple values with a single occurrence of the --scope and --audience options. We think that both approaches are equally usable. However, while multiple audiences can be passed as a space separated list, multiple scope values are passed as a comma separated list. This is inconsistent within the application. Furthermore, OIDC scopes are usually passed as a space separated list when passed as a single value. However, while htgettoken states that multiple scope values should be passed as a comma separated list, it is actually possible to pass them as a space separated list (as shown in the command above). We assume that a comma separated list will be split by htgettoken while a space separated list will not. However, since in OIDC scopes are passed as a space separated list to the OP, the space separated list of scopes passed to htgettoken does also work (but might be perceived as a single scope by htgettoken). The option -o /dev/stdout tells htgettoken to print the access token to the terminal, by default it is stored in a file located under /tmp (where other WLCG tools expect it).

As argued in subsection 2.3.3, storing vault tokens without encryption can lead to usability problems. From a security perspective tokens that are stored without encryption

should be as short-lived as possible. In practice this is a security-usability trade-off, where a shorter lifetime requires users to re-authenticate at the OP. While htgettoken has support for Kerberos integrated which can be used to easily obtain a new vault token without re-authenticating through OIDC, this is obviously not possible for non-Kerberos users.

Summary While oidc-agent offers good usability and nice features as a local agent, it is less suitable when access tokens need to be obtained from multiple machines, especially in the context of long running compute jobs.

htgettoken can also be easily used on the command line and offers integration with Kerberos. However, it was specifically developed for the WLCG community, their requirements, and policies. This means that it is less usable for other communities with other requirements. The biggest drawback in our opinion is that vault tokens are stored without encryption. This forces users to handle encryption themselves or use short-lived vault tokens. The latter forces users without Kerberos to re-authenticate fairly frequently.

mytoken fulfills all requirements of the project, in particular it can easily be used on multiple machines to obtain access tokens on the command line. Mytoken tokens can be copied for usage on another machine, just like vault tokens for htgettoken. In addition, mytoken offers transferring a mytoken with a transfer code, in case copy-paste is not suitable. The mytoken command line client offers good usability on the command line and handles the management of mytokens. This enables users to easily create multiple mytokens, each restricted to the particular use case. Mytoken's *restrictions* mechanism allows users to restrict their mytokens in a very flexible manner. Restrictions give users much more flexibility and freedom as it would be possible with OIDC tokens or htgettoken vault tokens. Users can easily create a mytoken for a particular compute job that will only be valid for the relevant time and offers obtaining access tokens with only the relevant scope and audience values. It is even possible to use a single mytoken to obtain access tokens with different scopes and audiences at different points in time, which allows modeling the complex requirements of long compute jobs.

6. Conclusion

In this chapter we give a summary over the work done throughout this thesis and present the future work that can be done to extend the mytoken service.

6.1. Summary

The goal of this thesis was to implement a solution for the long-running-jobs problem: When compute jobs are started with OpenID Connect (OIDC) access tokens but run for a long time, the access token used to start the job will eventually expire. Consequently, it cannot be used at the end of the job to store the results (and also not in between to access OIDC resources). Since it is not possible to request access tokens with an extensive lifetime from OpenID Providers (OPs), access tokens must be refreshed in between. Existing mechanisms offered by the OIDC protocol are insufficient. Also, existing software did not fully satisfy our requirements. We therefore designed and developed a central token service called mytoken that introduces the new token type "mytoken" which can be used across multiple machines to obtain OIDC access tokens. As part of this thesis we implemented the server as well as a command line client.

The mytoken service solves the long-running-jobs problem through mytokens. These mytokens can be used to obtain additional access tokens, similarly to OIDC refresh tokens. However, unlike refresh tokens, mytokens can be easily passed around, so that they can be used on different machines. To balance the security with usefulness we introduced the concepts of *capabilities* and *restrictions*, which both can be controlled by the user so they can create mytokens that are capable of doing exactly what is required without opening room for possible attacks more than necessary. Capabilities describe what a mytoken can be used for, e.g. to obtain access tokens, while restrictions are used to restrict a mytoken as much as possible. The different dimensions include:

Time When the mytoken can be used for the first and last time.

Location From which ip addresses / networks, but also countries, the mytoken can be used.

OIDC The scopes and audiences that access tokens obtained from the mytoken may have.

Number of Usages How often a mytoken can be used.

Multiple restriction clauses may be used to allow multiple "usage settings", e.g. to obtain an access token with one set of scopes at the beginning of a compute job and another set of scopes at the end of the job. The combination of easy mytoken creation from an existing token and the power of restrictions allows users to create mytokens often and as precise

as required. A restricted mytoken can be passed to other applications without worrying that it might be misused, because it can be restricted to only allow specified operations. This makes the mytoken service a useful tool for token based scenarios in which security and longer-term authorization are required.

We published the mytoken server[1] and client[2] under the MIT license. A demonstration instance[3] is available for testing the service. Packages for multiple operating systems are available for a self-hosted operation by communities that require it.

6.2. Future Work

The developed mytoken service and client are fully usable, but currently do not use their full potential. Different extensions are possible and we want to describe some possible future work.

Extending Grant Types for Mytoken Endpoint Currently, there are two ways for obtaining mytokens from the server: either using the authorization code flow or from an existing mytoken. While the former is required to be done at least once, it will be used less. The latter is likely to be the main way for obtaining additional mytokens. However, we already designed mytoken so that additional ways for obtaining mytokens can be implemented. These ways are called grant types. In the following we describe three additional grant types that can be implemented.

Kerberos While we do not see Kerberos integration in our use cases, there are communities which heavily use Kerberos. These communities can greatly benefit from a Kerberos grant type that can be used to obtain mytokens. The challenge for this grant type will be to implement it in a way that the encryption of refresh tokens as described in section 4.1 is still retained.

Access Token An access token grant type could be used to obtain a mytoken from an OIDC access token. Since the server then only has the access token and no additional information, it cannot access an encrypted refresh token[4]; therefore, this grant type would need to perform a token exchange flow to obtain a refresh token from the OP which then can be linked to the new mytoken. This grant type would enable users to obtain an initial mytoken (root of a mytoken subtoken tree) without performing an OIDC flow. Because an attacker must not be able to use any access token it can get its hand on to obtain a valid mytoken from it, this grant type must be explicitly enabled by the user and only accept access token that are meant to be used with the mytoken server, i.e. they must have the mytoken server listed as a valid audience. It would also be possible to require a specific scope for such an access token, e.g. `create_mytoken`.

[1] https://github.com/oidc-mytoken/server/releases
[2] https://github.com/oidc-mytoken/client/releases
[3] https://mytoken.data.kit.edu
[4] At least not if the refresh token is encrypted with the desired property from section 4.1

Private Key JWT With this grant type users can upload a cryptographic public key and use the private key to do cryptographic operations as authentication. This way they prove their identity by proving the possession of the private key. One popular approach for this is that the user signs its request. This could be done by using the JSON request as the payload for a JSON Web Token (JWT) and instead of the JSON request the signed JWT is sent. However, this approach will hardly work with the encryption of refresh tokens we already referred to in the other two grant types. The problem is that this only authenticates the user, i.e. the mytoken server can be confident that it is actually the right user that sends the request, but the request does not give the server any information that can be used to decrypt the refresh token (at least no information that is not already known by the server). However, a similar approach can be used with encryption instead of signing. This approach would include a challenge-response-protocol where the client solves a challenge from the server to prove the possession of the private key. Such a protocol can then be used for decrypting the stored refresh token. Furthermore, the grant type can be designed so that the client does not learn the refresh token and that a transcript of the communication cannot be used for replay attacks.

We envision that these grant types must be explicitly enabled by the user. For some of these, this is a requirement, because additional information must be provided by the user to enable them (i.e. account linking for Kerberos, uploading the public key for private key JWT). So that users can enable additional grant types and configure them, an additional endpoint is needed. We would call such an endpoint *user settings* endpoint and it can be used to adapt all sorts of user settings, like enabling additional grants, but it could also be used to set restriction and capability defaults per user.

Rotating Mytokens To mitigate the risk of stolen mytokens it would be possible to implement rotating mytokens. This feature would work similar to rotating refresh tokens. Each mytoken can only be used once, and whenever a mytoken is used the current one is invalidated and instead a new mytoken is created and returned along with the response. This feature could be enabled by users on a per token base, so that it is possible to create some mytokens with rotation and some without. This is useful because a mytoken might be passed to an application that cannot handle the rotation. We also envision that the rotation can be controlled separately for access token requests and for other requests.

Templates and Profiles for the Mytoken Client The mytoken client does currently not have a default value for restrictions. This is mainly because it is difficult to set a default that is suitable for all users. However, it is clearly desirable to have a sensible default value instead of returning unrestricted mytokens by default. To achieve this, templates could be used. Another motivation for templates (and profiles) is the fact that restrictions, despite their flexibility, can also become quite complex and that this complexity might not be appropriate for all users. To still enable these users to use the full power of restrictions without much configuration effort templates can be used. The last aspect of motivating templates is that they can reduce the length of a mytoken command, so that commands become cleaner.

We envision templates to be predefined restrictions that live as JSON in a file. These templates can then easily be used by referencing their name. This approach enables communities to provide templates to users, so that they can install the templates and just use it without much effort. We envision that there are two locations where templates are stored: a system wide location like /etc/mytoken/capabilities.d/ and a user wide one like ~/.config/mytoken/capabilities.d/. This enables the easy distribution of templates, but also allows users to overwrite templates easily and to write their own templates.

With the template approach we envision that a mytoken with an expiration restriction of one day could be requested with any of the following commands:

```
mytoken MT --exp=+1d                  # using the special command line options
mytoken MT --restrictions='{"exp":"+1d"}'    # using json
mytoken MT --restrictions=/tmp/restriction   # using json in a file
mytoken MT --restrictions=@one-day           # using the 'one-day' template
```

The same template mechanism could then also be used with capabilities. Then capabilities and restrictions can be combined in a *profile*. Profiles could be a simple combination of capabilities and restrictions (which can be specified directly or use a template in the profile file). This would allow communities to not only provide templates but also profiles to their users, so they only have to provide the profile name and do not need to care about capabilities or restrictions.

Standardization To be useful, the mechanisms introduced with mytoken must be well documented, thoroughly discussed, and published. The first step for achieving this has be done with this thesis. One possible way to continue this work is a standardization document, for example as an RFC.

Bibliography

[1] Apache. *core - Apache HTTP Server Version 2.4*. 2020. URL: http://httpd.apache.org/docs/2.4/mod/core.html#limitrequestfieldsize (visited on 02/13/2021).

[2] GÉANT Association. *Worldwide LHC Computing Grid - AARC*. 2017. URL: https://aarc-project.eu/aarc-in-action/wlcg/ (visited on 04/15/2021).

[3] Vittorio Bertocci. *JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens*. Internet-Draft draft-ietf-oauth-access-token-jwt-12. IETF Secretariat, Mar. 2021. URL: http://www.ietf.org/internet-drafts/draft-ietf-oauth-access-token-jwt-12.txt.

[4] Isabel Campos, Diana Gudu, and Marcus Hardt. "HPC AAI Integration". EGI Conference 2020. Nov. 2020. URL: http://marcus.hardt-it.de/pam_ssh/.

[5] Andrea Ceccanti. *INDIGO IAM for wlcg*. 2021. URL: https://wlcg.cloud.cnaf.infn.it/.

[6] Andrea Ceccanti. *OAuth Token exchange support - iam*. 2018. URL: https://indigo-dc.gitbook.io/iam/user-guide/oauth_token_exchange (visited on 03/29/2021).

[7] CERN. *Certificates | WLCG*. URL: https://wlcg.web.cern.ch/certificates (visited on 04/15/2021).

[8] CERN. *Home | CERN*. 2021. URL: https://home.cern/ (visited on 02/03/2021).

[9] CERN. *Welcome | Worldwide LHC Computing Grid*. 2021. URL: https://wlcg-public.web.cern.ch/ (visited on 02/03/2021).

[10] European Commission. *Horizon Europe | European Commission*. 2021. URL: https://ec.europa.eu/info/horizon-europe_en (visited on 02/03/2021).

[11] European Commission. *What is Horizon 2020? | Horizon 2020*. 2017. URL: https://ec.europa.eu/programmes/horizon2020/en/what-horizon-2020 (visited on 02/03/2021).

[12] D. Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. RFC Editor, May 2008. URL: http://www.rfc-editor.org/rfc/rfc5280.txt.

[13] W. Denniss et al. *OAuth 2.0 Device Authorization Grant*. RFC 8628. RFC Editor, Aug. 2019.

[14] DFN. *DFN-AAI*. 2019. URL: https://www.dfn.de/dienstleistungen/dfnaai/ (visited on 04/21/2021).

[15] Dave Dykstra, Mine Altunay, and Jeny Teheran. "Secure Command Line Solution for Token-based Authentication". May 2021. URL: https://github.com/fermitools/htgettoken/files/6063416/CHEP21_Paper_Htgettoken.pdf.

[16] David Dykstra. *fermitools/htgettoken: Gets OIDC authentication tokens for High Throughput Computing via a Hashicorp vault server*. 2020. URL: https://github.com/fermitools/htgettoken.

[17] eduGAIN. *eduGAIN - enabling wordwide access*. URL: https://edugain.org/ (visited on 04/21/2021).

[18] eduGAIN. *Key concepts - eduGAIN*. URL: https://edugain.org/about-edugain/key-concepts/ (visited on 04/21/2021).

[19] EGI Foundation. *EGI FedCloud*. 2021. URL: https://aai.egi.eu/fedcloud/index.php (visited on 04/16/2021).

[20] EOSC-hub. *EOSC Hub*. 2020. URL: https://www.eosc-hub.eu/ (visited on 02/03/2021).

[21] EOSC-synergy. *EOSC synergy - Building capacity, developing capacity*. 2020. URL: https://www.eosc-synergy.eu/ (visited on 02/03/2021).

[22] Fiber. *Fiber*. 2021. URL: https://gofiber.io/ (visited on 04/07/2021).

[23] EGI Foundation. *EGI | EGI-ACE: Advanced Computing for EOSC*. 2020. URL: https://www.egi.eu/projects/egi-ace/ (visited on 02/03/2021).

[24] OpenID Foundation. *Certified OpenID Connect Implementations | OpenID*. 2021. URL: https://openid.net/developers/certified/ (visited on 04/06/2021).

[25] Go Supported by Google. *The Go Programming Language*. URL: https://golang.org/ (visited on 04/20/2021).

[26] D. Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. RFC Editor, Oct. 2012. URL: http://www.rfc-editor.org/rfc/rfc6749.txt.

[27] HarshiCorp. *Vault by HashiCorp*. 2021. URL: https://www.vaultproject.io/ (visited on 03/05/2021).

[28] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. http://www.rfc-editor.org/rfc/rfc7519.txt. RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7519.txt.

[29] M. Jones and D. Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. RFC 6750. RFC Editor, Oct. 2012. URL: http://www.rfc-editor.org/rfc/rfc6750.txt.

[30] M. Jones et al. *OAuth 2.0 Token Exchange*. RFC 8693. RFC Editor, Jan. 2020.

[31] Karlsruhe Institute of Technology. *bwIDM - Über bwIDM*. URL: https://www.bwidm.de/ (visited on 04/16/2021).

[32] T. Lodderstedt, S. Dronia, and M. Scurtescu. *OAuth 2.0 Token Revocation*. RFC 7009. RFC Editor, Aug. 2013.

[33] Andy Manoske. *How does vault encrypt data?* 2018. URL: https://www.hashicorp.com/resources/how-does-vault-encrypt-data (visited on 04/15/2021).

[34] AARC Community members and AppInt members. *AARC Blueprint Architecture 2019 (AARC-G045)*. Nov. 2019. DOI: 10.5281/zenodo.3672785. URL: https://doi.org/10.5281/zenodo.3672785.

[35] NGINX. *Module $ngx_http_core_module$*. 2020. URL: http://nginx.org/en/docs/http/ngx_http_core_module.html#large_client_header_buffers (visited on 02/13/2021).

[36] OASIS. *Security Assertion Markup Language (SAML) V2.0 Technical Overview*. Tech. rep. May 2008. URL: http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html.

[37] Keisuke Okamura. "Interdisciplinarity revisited: evidence for research impact and dynamism". In: *Palgrave Communications* 5.1 (Nov. 2019), p. 141. ISSN: 2055-1045. DOI: 10.1057/s41599-019-0352-4. URL: https://doi.org/10.1057/s41599-019-0352-4.

[38] Codership Oy. *Products | Gerla Cluster for MySQL*. 2019. URL: https://galeracluster.com/products/ (visited on 03/09/2021).

[39] Ping Identity. *SAML 2.0: How It Works*. 2015. URL: https://www.pingidentity.com/en/resources/client-library/articles/saml.html (visited on 04/21/2021).

[40] AARC project. *AARC*. 2017. URL: https://aarc-project.eu/ (visited on 02/03/2021).

[41] N. Sakimura, J. Bradley, and M. Jones. *OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1*. Tech. rep. Nov. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html.

[42] N. Sakimura et al. *OpenID Connect Core 1.0 incorporating errata set 1*. Tech. rep. Nov. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html.

[43] Nat Sakimura. *Apple Successfully Implements OpenID Connect with Sign In with Apple | OpenID*. 2021. URL: https://openid.net/2019/09/30/apple-successfully-implements-openid-connect-with-sign-in-with-apple/ (visited on 04/06/2021).

[44] Hannah Short et al. *WLCG Authorisation; from X.509 to Tokens*. Tech. rep. Nov. 2019. URL: https://indico.cern.ch/event/773049/contributions/3473383/.

[45] Klaas Wierenga et al. *EOSC Authentication and Authorization Infrastructure (AAI) : Report from the EOSC Executive Board Working Group (WG) Architecture AAI Task Force (TF)*. 46.21.02; LK 01. European Commission (EU), Jan. 2021. 24 pp. ISBN: 978-92-76-28113-9. DOI: 10.2777/8702.

[46] Gabriel Zachmann. *Agent forwarding - oidc-agent*. 2020. URL: https://indigo-dc.gitbook.io/oidc-agent/configuration/forwarding.

[47] Gabriel Zachmann. *indigo-dc/oidc-agent: oidc-agent for managing OpenID Connect tokens on the command line*. 2020. URL: https://github.com/indigo-dc/oidc-agent.

[48] Gabriel Zachmann. "OIDC-Agent: Managing OpenID Connect Tokens on the Command Line". In: *SKILL 2018 - Studierendenkonferenz Informatik*. Ed. by Michael Becker. Bonn: Gesellschaft für Informatik e.V., 2018, pp. 11–21.

A. Appendix

OIDC Userinfo Response


```
{
  "fieldOfStudyText": "Informatik",
  "sub": "c759d997-24c9-43a3-bb27-a0ce47bc4e03",
  "address": {},
  "email": "example@student.kit.edu",
  "displayName": "Zachmann, Gabriel",
  "givenName": "Gabriel",
  "preferred_username": "example",
  "given_name": "Gabriel",
  "eduperson_principal_name": "example@student.kit.edu",
  "eduperson_entitlement": [],
  "upn": "example@student.kit.edu",
  "name": "Gabriel Zachmann",
  "eduperson_scoped_affiliation": [
    "member@kit.edu",
    "student@kit.edu"
  ],
  "sn": "Zachmann",
  "family_name": "Zachmann"
}
```


Listing A.1: Example response from the userinfo endpoint

Mytoken Example Responses


```
{
  "issuer": "https://mytoken.data.kit.edu/",
  "access_token_endpoint": "https://mytoken.data.kit.edu/api/v0/token/access",
  "mytoken_endpoint": "https://mytoken.data.kit.edu/api/v0/token/my",
  "tokeninfo_endpoint": "https://mytoken.data.kit.edu/api/v0/tokeninfo",
  "revocation_endpoint": "https://mytoken.data.kit.edu/api/v0/token/revoke",
  "token_transfer_endpoint": "https://mytoken.data.kit.edu/api/v0/token/transfer",
  "jwks_uri": "https://mytoken.data.kit.edu/jwks",
  "providers_supported": [{
      "issuer": "https://wlcg.cloud.cnaf.infn.it/",
      "scopes_supported": [
        "openid", "profile", "email", "storage.create:/", "storage.read:/",
          "storage.modify:/", "wlcg", "wlcg.groups",
          "eduperson_scoped_affiliation", "eduperson_entitlement"
      ]
    }, {
      "issuer": "https://aai-dev.egi.eu/oidc/",
      "scopes_supported": [
        "openid", "profile", "email", "eduperson_entitlement",
          "eduperson_scoped_affiliation", "eduperson_unique_id",
          "cert_entitlement", "orcid", "ssh_public_key"
      ]
    }],
  "token_signing_alg_value": "ES512",
  "tokeninfo_endpoint_actions_supported": [
    "introspect", "event_history", "subtoken_tree", "list_mytokens"
  ],
  "access_token_endpoint_grant_types_supported": ["mytoken"],
  "mytoken_endpoint_grant_types_supported": [
    "oidc_flow", "mytoken", "transfer_code", "polling_code"
  ],
  "mytoken_endpoint_oidc_flows_supported": ["authorization_code"],
  "response_types_supported": [
    "token", "short_token", "transfer_code"
  ],
  "service_documentation": "https://mytoken-docs.data.kit.edu/",
  "version": "0.2.0"
}
```


Listing A.2: Example response from the mytoken configuration endpoint

Fiber Benchmark

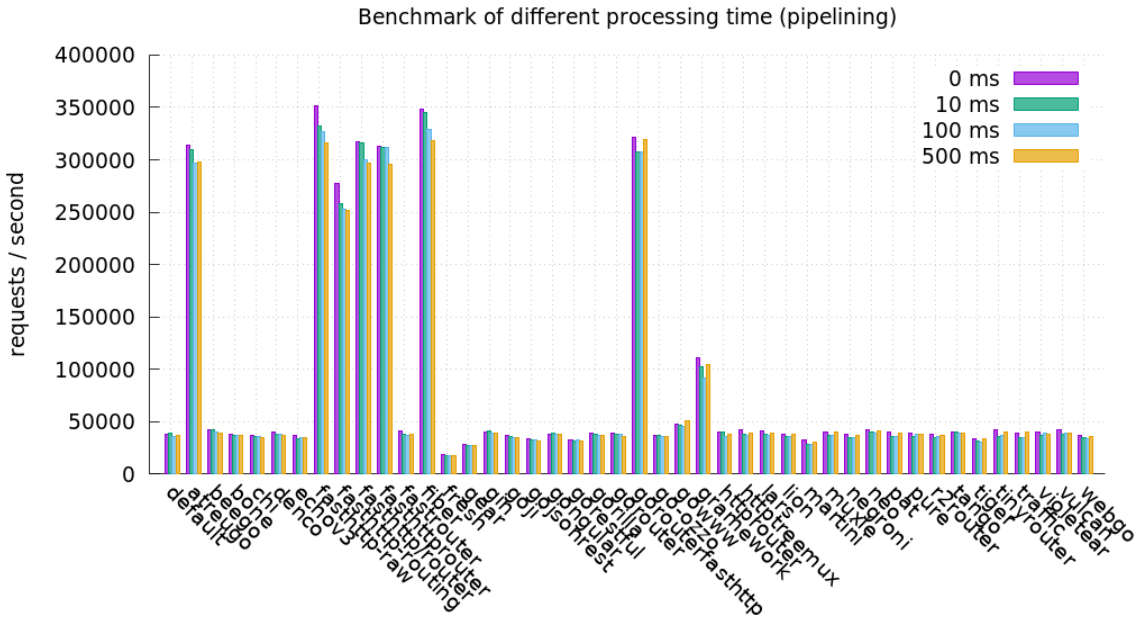


Figure A.1.: Benchmark of different Go web frameworks [22]