BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FEDCM API INTEGRATION INTO KEYCLOAK

INTEGRACE FEDCM API DO SYSTÉMU KEYCLOAK

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ŠIMON VACEK**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. RADEK BURGET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

156111

Institut:	Department of Information Systems (DIFS)
Student:	**Vacek Šimon**
Programme:	Information Technology
Title:	**FedCM API Integration into Keycloak**
Category:	Information Systems
Academic year:	2023/24

Assignment:

1. Learn about the architecture and use of KeyCloak for centralized identity and access management.
2. Study current practices and protocols for implementing single sign-on in web applications. Focus on the emerging FedCM standard.
3. Based on your consultations with RedHat, propose a way to extend KeyCloak to support the FedCM API.
4. Implement the proposed solution using appropriate technologies.
5. Validate the functionality of the designed solution in appropriate applications.
6. Evaluate the achieved results.

Literature:

- Bertocci, V.: OAuth2 and OpenID Connect: The Professional Guide, Okta, Inc., 2022
- Dále dle pokynů vedoucího.

Requirements for the semestral defence:
Items 1 to 3

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Burget Radek, doc. Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	30.10.2023

Abstract

Because of security concerns, trustworthiness, and ongoing privacy-oriented changes, third-party cookies are to be phased out in web browsers. These play a key role in federating user identities in single sign-on applications, so a unified solution with a focus on preserving privacy is being developed. The Federated Credential Management API is the proposed solution; at this time, it is not yet standardized. This thesis deals with implementing the specification of this API to a Cloud Native Computing Foundation incubating project Keycloak. It is an open source single-sign-on application with Red Hat as the main contributor. The OpenID Connect and OAuth 2.0 are discussed together with how they work together with FedCM and all of its extensions. The result is implemented as a service provider interface extension to Keycloak.

Abstrakt

Kvůli obavám o bezpečnost, důvěryhodnost a kvůli probíhajícím změnám v bezpečnosti dochází ve webobých prohlížečích k zákazu cookies třetích stran. Ty hrají klíčovou roli při federování identit uživatelů v aplikacích jednotného přihlášení, proto se vyvíjí jednotné řešení s důrazem na ochranu soukromí. Federated Credential Management API je v tuto chvíli navržené a prozatím nestandardizované řešení. Tato práce se zabývá implementací specifikace tohoto API do Cloud Native Computing Foundation inkubačního projektu Keycloak. Jde o aplikaci jednotného přihlášení s otevřeným zdrojovým kódem, jejíž hlavní přispěvatel je firma Red Hat. OpenID Connect a OAuth 2.0 jsou popsány společně s tím jak fungují dohromady s FedCM a všemi jeho rozšířeními. Výsledek je implementován jako rozšíření rozhraní poskytovatele služeb Keycloaku.

Keywords

Keycloak, single sign-on, third-party cookies, FedCM, federated identity, user identity, social login

Klíčová slova

Keycloak, jednotné přihlášení, cookies třetích stran, FedCM, federovaná identita, identita uživatele, sociální přihlášení

Reference

VACEK, Šimon. *FEDCM API INTEGRATION INTO KEYCLOAK*. Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Radek Burget, Ph.D.

Rozšířený abstrakt

Při brouzdání internetovými stránkami je běžné, že se uživatel musí přihlásit, aby získal přístup k veškerým jejich funkcím. Na základě informací o uživateli může stránka, nebo klient, zobrazovat relevantnější informace. Např. webový portál s databází filmů a seriálů IMDb po přihlášení umožní uživateli hodnotit zhlédnuté filmy, vytvářet seznamy pro pozdější zhlédnutí a doporučovat další obsah. Počet účtů, kterými běžný člověk disponuje, je ale značný a pro každý z nich by měl mít jedinečné a bezpečné heslo. Proto tyto portály umožňují použít účet od některého z poskytovatelů identit. Např. Google, nebo Facebook nabízí takovouto službu pod tlačítkem "Přihlásit se přes ...". Tuto službu poskytuje i Cloud Native Computing Foundation inkubační projekt Keycloak. Jedná se o systém jednotného přihlášení (SSO), který umožňuje mnoho věcí, například jednotnou správu identit a snadné zabezpečení aplikací pro velké i malé firmy.

Obdobné aplikace pro svojí funkčnost používají malé soubory dat uložené v prohlížeči, označované jako cookies. Tyto cookies se dělí podle svého původu a mohou být zneužívány pro sledování pohybu uživatele napříč webovými stránkami. Proto se tyto cookies patřící do jiných domén v prohlížečích blokují. Aby bylo zachováno fungování aplikací jako je Keycloak, iniciativa Privacy Sandbox for the Web přichází s návrhem jak tyto cookies využít bezpečným způsobem a umožnit přihlášení skrze třetí stranu - poskytovatele identit. Tato práce přibližuje principy serveru Keycloak a protokoly, které používá pro autentizaci a autorizaci. A také se zabývá aktuálním řešením v případě blokovaných cookies. Zmíněné protokoly jsou OpenID Connect a OAuth 2.0. U těchto protokolů jsou popsány jejich základní principy a datové struktury.

Dále je do hloubky rozebráno ono řešení – aplikační programové rozhraní pro správu federovaného pověření (Federated Credential Management API, zkráceně FedCM API). Specifikace tohoto rozhraní je v době psaní této práce nestandardizovaná a otevřená pro případné změny. Zmíněny jsou proto i vlastnosti, které jsou prozatím pouze prototypizovány a nejsou plně adoptovány. Je popsáno nejen rozhraní, které se implementuje na straně Keycloaku, ale také rozhraní prohlížeče, který hraje klíčovou roli při předávání identity uživatele klientské aplikace. Velký důraz je přitom kladen na potvrzení uživatele, který musí odsouhlasit toto předání.

Cílem této práce je FedCM API integrovat do projektu Keycloak, demonstrovat jeho aplikovatelnost a kriticky zhodnotit jeho použití v budoucnu. Výsledné řešení je navrženo jako rozšíření projektu za využití rozhraní poskytovatele služeb (Service Provider Interface). To umožňuje dynamicky přidávat funkcionalitu Keycloak serveru bez zásahu do kódu aplikace. Hotová implementace pokrývá základní požadavky specifikace a umožní klientské aplikaci vytvořit žeton (anglicky též token), který aplikace může využít pro identifikování uživatele a přistoupení uživatelově zdrojům. Toto řešení není pro projekt takovýchto rozměrů s důrazem na bezpečnost zcela dostačující. Jako ověření konceptu, nabízí ale důležitý bod, odkud lze dále vycházet při aplikaci komplexnejších požadavků a např. použití protokolu SAML.

Federated Credential Management je zhodnocen jako odvážná iniciativa, kterou bude nezbytné adoptovat. Rozhodnutí nevázat se na žádné jiné protokoly může ale v budoucnu způsobit překážky, které ztíží plnou integraci dalšími poskytovateli identit a může způsobit přepisování již existujících řešení.

FEDCM API INTEGRATION INTO KEYCLOAK

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of doc. Ing. Radek Burget, Ph.D. The supplementary information was provided by Jon Koops and Stian Thorgersen. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Šimon Vacek
May 7, 2024

Acknowledgements

I would like to thank doc. Ing. Radek Burget, Ph.D., for supervising my thesis. Many thanks also belong to Jon Koops and Stian Thorgersen from Red Hat for giving me this opportunity, consulting my work, and their leadership. My appreciation goes as well to my colleagues Michal Hajas, Hynek Mlnařík, and Martin Bartoš for helping me understand the sometimes confusing Keycloak.

Contents

List of Figures

Chapter 1

Introduction

Every website, application, or service seems to require an account to use their service. The number of passwords needed to remember is too significant, so many developers have decided to use some form of social login. The famous *„Sign in with Facebook“* or Google or other social media has become very popular. This brings the benefit that there is no need to create an account or an identity for every online app used. The identity is simply federated to another application. Unfortunately, the underlying mechanisms are abused to track users without their consent or complete understanding.

To make the identity federation work, identity providers, for example, Facebook or Google, use small chunks of data stored in the browser called cookies to initiate login and hold information about the authentication status. In this context, the cookies are third-party cookies because they belong to a different website than the website trying to establish the user's identity. Starting the year 2024 with the initiative **Privacy Sandbox for the Web**, third-party cookies will be phased out. Using them at all will not be possible by the end of the year. To preserve applications relying on their use, the **Federated Credential Management API** is introduced as a solution.

The Keycloak project is an open-source server that handles authentication, authorization, user management, and single sign-on (SSO). That means it allows the user to log in only once and be authenticated across multiple applications registered in Keycloak. That way, a user, for example, logs in to Keycloak and then access their Google account and the company intranet. With the already ongoing cookie restrictions, Keycloak had to adapt and provide workarounds. These are unsustainable and worsen the user experience. This affects namely the Javascript library, which could hopefully be deprecated altogether in the future.

Federated Credential Management API introduces a mechanism that is a part of the internet browser and an identity provider. It places the browser as a trustworthy party in the communication that negotiates the identity federation from the identity provider to the client website, the relying party. As of the writing and publishing of this thesis, FedCM API is not yet standardized and is open for change. The outlined specification is open for developers to adopt the technology and prototype and give feedback on the missing or not working pieces in the specification.

In chapter 2, the thesis describes the underlying protocols OpenID Connect and OAuth 2.0 used in Keycloak for authenticating and authorizing. It is important to understand particularly the issued tokens bearing the actual credentials. The chapter 3 discusses Keycloak and its concepts, such as realms. Part of the chapter also mentions the current solution to the third-party cookies issue and points out two important features in danger for

the phase-out. This text's core is the FedCM specification discussed in chapter 4. Not only the mandatory part of the specification for both the web browser and the identity provider is examined, but experimental features like the dynamic sign-in flow are also examined. Finally, chapter 5 outlines the proposal for how this API could be integrated into Keycloak using the Service Provider Interface. The results of the implementation process and plans for the Keycloak FedCM extension's future are in the last chapter 6.

Chapter 2

OpenID Connect and OAuth 2.0

Before delving into the Federated Credential Management API, the context of Keycloak and the protocols it uses for authentication and authorization must first be explained. Although it is more sensible to first approach Keycloak itself, the Keycloak chapter 3 already needs and mentions parts of these protocols, so these need to be explored first. This thesis discusses the integration of FedCM[1] API into Keycloak, which utilizes two protocols for authentication and authorization: SAML[2] and OIDC[3]. For the sake of simplicity, I have decided to go with only OIDC as it is a newer standard working with JSON[4] rather than XML[5] files.

Keycloak serves as an authentication and authorization server. It holds information about a user and controls access by other applications to this information (resources) and any other resources selected to be protected. Therefore, Keycloak can be an authentication server and a resource server. The resource owner is an end-user who can grant permission to third-party applications to access these resources on their behalf. The OAuth 2.0 Authorization Framework was designed as a secure way to do just this. So, the third-party application does not need to use the end user's credentials and store them; it introduces a new set of credentials, tokens, issued by the authorization server, which are used to access the resource owner's protected resources. It builds an authorization layer.

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 [18]. Essentially, it extends the OAUTH 2.0 specification to provide authentication and information about the user. There are three actors referred to the specification:

- **End-User**, a user
- **OpenId Provider** (OP), the authentication server
- **Relying party** (RP), a client, an application asking for authentication

The ultimate goal of an RP is to obtain tokens from the OP, which they can use to access protected resources and provide personalized services. These tokens include an **Access token**, a **Refresh token**, and an **ID token**.

[1] Not to be confused with https://fedcm.org/ — Federation of Christian Ministries
[2] Security Assertion Markup Language
[3] OpenID Connect
[4] JavaScript Object Notation
[5] Extensible Markup Language

2.1 ID token

According to the OpenID Connect Core, the ID Token is a security token that contains Claims about the Authentication of an End-User by an Authorization Server when using a Client and potentially other requested Claims. The ID Token is represented as a JSON Web Token (JWT)[6] [18]. Claims are pieces of information included in a token. It could be an email, name, user ID, and information needed for authentication, like the URL of the token's issuer, the timestamp when it was issued, and the timestamp when this token expires. This token establishes the end-user's identity and must be signed using JWS[7] (JSON Web Signatures). The latter two tokens are used to authorize an RP repeatedly.

2.2 Access token

An Access token is defined in the OAuth 2.0 specification as a string representing an authorization issued to the client [8]. The specification alone does not describe what the token should look like other than it is usually opaque to the client (that means the client is not meant to understand the contents). This token authorizes the client to access protected resources; the Client does not need to prove its legitimacy when using it.

Later, a **Bearer token** defined in RFC6750[8] was introduced as a type of token that could be used as an Access token. The specification requires using TLS[9] when making requests with bearer tokens. The client must validate the TLS certificate chain [9]. This eliminates some of the potential security concerns.

Because the OIDC protocol is described in this thesis in the context of the Keycloak project, it is worth mentioning that Keycloak uses JWTs for Access Tokens. This is because, as a widely used standard format, it does not pose much trouble to integrate. The token is based on JSON, so it can be easily parsed and worked with in any programming language [20].

2.3 Refresh token

The refresh token serves a special purpose: obtain a new access token and, optionally, a new ID token from the authorization server (Keycloak). Because of security concerns, the access token has a time assigned to it (usually a short period) when it expires, and resource servers must no longer accept it. A refresh token has a longer lifetime. The refresh token is issued optionally when requested (`refresh_token` value is present in the `grant_type` parameter in the refresh request [18]) and is included together with an access token. It is to be used only for the authorization server and never a resource server. It is, again, usually opaque to the client application.

[6] https://www.rfc-editor.org/rfc/rfc7519
[7] https://www.rfc-editor.org/rfc/rfc7515
[8] https://www.rfc-editor.org/rfc/rfc6750
[9] Transport Layer Security

2.4 OIDC Authentication flows

There are three ways to authenticate with OIDC. Two of these flows are taken from OAuth 2.0 specification and extended. A developer decides which flow is the most suitable for their client (the RP) based on its capabilities. Based on the client's ability to authenticate securely with the authorization server, clients are divided into two groups [8]:

- **Confidential**: this client is capable of secure client authentication
- **Public**: this client is not capable of secure client authentication, typically a native application or a web-browser-based application

Client authentication is recommended and is typically done during client registration with the OP. The client registration can be done in Keycloak in the administration console via an HTML form. The admin provides a list of valid redirect URIs (used to redirect a user back to the client after authenticating them) and chooses a client type. If the type is `confidential` (in Keycloak, the Client authentication is turned on), the typical client authentication is with a chosen client id and a generated client secret.

2.4.1 Authorization code flow

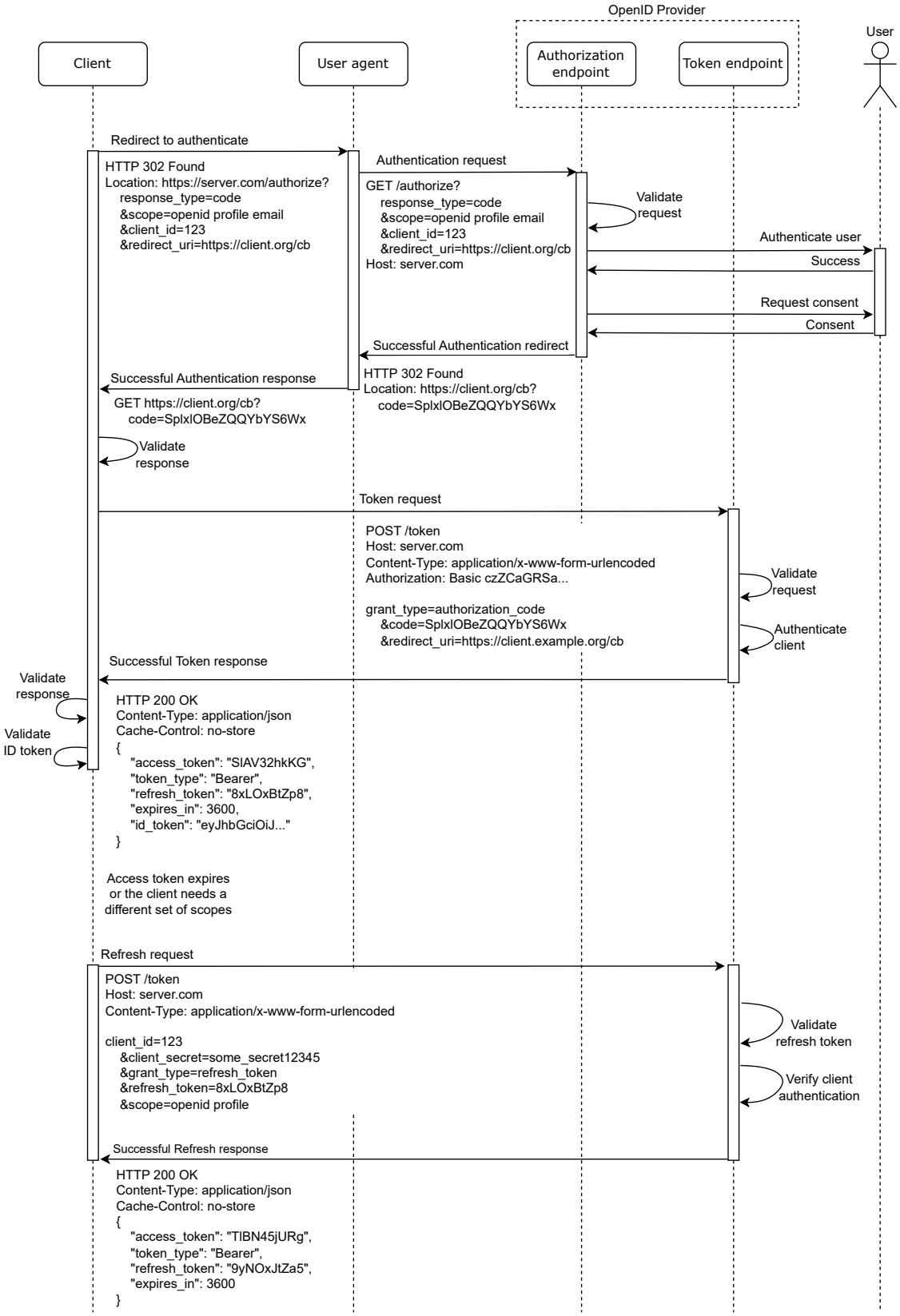
The authorization code flow is the most secure flow OIDC offers and is redirection-based. The clients using it are typically `confidential`, and it is recommended to have some client authentication established. It starts with a client sending an authentication request, which triggers the user agent to redirect to the authorization endpoint of the OP. The OP's authorization server validates the request with its parameters and optionally authenticates the client at this point. The server authenticates the end-user and gathers their consent to share the requested claims with the RP.

If successful, the end-user is redirected back to the client using a URI defined in the `redirect_uri` parameter in the authentication request, which must also match the list of redirect URIs set during client registration. The successful response contains an authorization code in the `code` parameter. With this very short-lasting code and a redirect URI (the same as the one sent to the authorization endpoint) in the parameters, the client sends a token request directly to the OP's token endpoint. If the client type is confidential, it must be authenticated. Another series of checks is performed, including validating the authorization code and checking if it was issued to the authenticated client or not used before. The client is then issued an ID token, an access token, and optionally a refresh token. These are returned straight to the client, which validates the response, the ID token, and the access token.[10] For these validations, a `nonce` value is interesting.

The advantage of this flow is the security, the issuance of all three tokens, and the fact that the tokens are sent directly to the client, eliminating their exposure to the user agent. An example of a successful authorization code flow is in figure 2.1.

[10] More can be found in sections 3.1.3.7. and 3.1.3.8. [18].

Figure 2.1: Example of a successful **authorization code flow** authentication with client credentials and a refresh request. HTTP requests and responses are taken from the OIDC specification [18].



2.4.2 Implicit flow

This flow is best suited for clients implemented in a browser that cannot securely store client credentials. For this type of client, the `public` value is reserved. They are typically implemented using a scripting language such as Javascript. The Authorization Server does not perform Client Authentication.

The implicit flow is similar to the authorization code flow, with some exceptions. The token endpoint is not utilized at all, and an ID token and an access token are issued straight from the authorization endpoint. Obtaining a refresh token is impossible, so once the tokens expire, the client must perform the entire flow again.

The authentication starts with a request to the authorization endpoint of the OP with a `redirect_uri` that must match the registered redirect uris at the OP and a required `nonce` parameter. These two parameters were mentioned in 2.4.1. The `redirect_uri` defines an RP's URI the OP redirects to with an authentication response (be it a success or an error). The `nonce` is a string value that associates a client session with an ID token. This value can be used only once, which serves as a protection against replay attacks [18]. The OP validates the request, authenticates the end-user, and acquires their consent. If everything is in order, the user agent is redirected back to the client with an ID token and optionally an access token if the response mode parameter was set to `id_token token`; all response modes and their relation to issued tokens are described in [16].

This flow does not provide the same security as the authorization code flow, as the tokens are exposed to the end-user and applications with access to the user agent. Still, it is quicker and does not require as many requests to the OP when authenticating.

Figure 2.2 displays communication in a successful Implicit flow.

2.4.3 Hybrid flow

As the name suggests, the hybrid flow combines the previous two flows. The client sends an authentication request and receives an authorization code and either an ID token or an access token or both, depending on the `response_type` values in the request. The authorization code should be used on the token endpoint in exchange for an access token. The OP does not have to invalidate the previously issued access token, so it is possible to have more than one access token valid. This was also possible with the authorization code flow. This feature is usually used to request a token with a different scope. See figure 2.3 for a hybrid flow communication example.

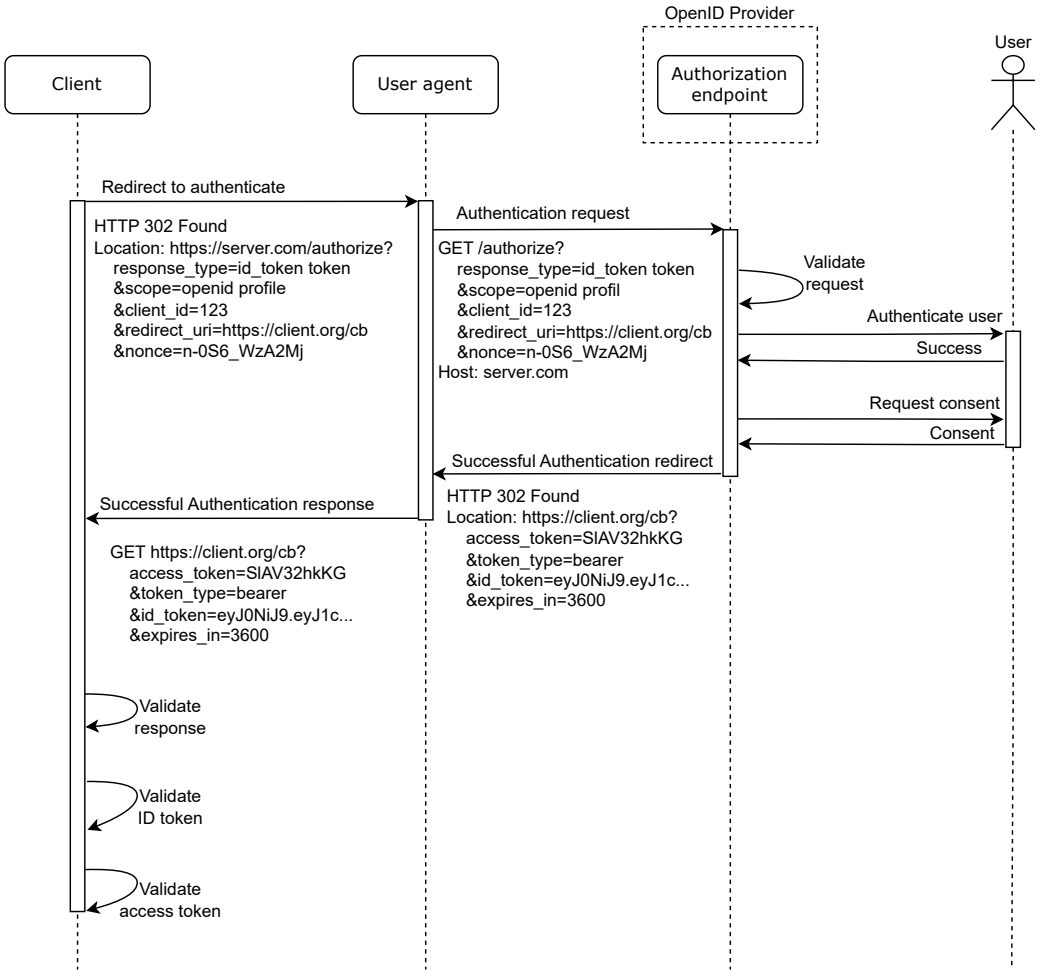


Figure 2.2: Example of a successful **implicit flow** authentication with an `id_token token` response type, which issues both an ID token and an access token. The access token is optionally validated. HTTP requests and responses are taken from the OIDC specification [18].

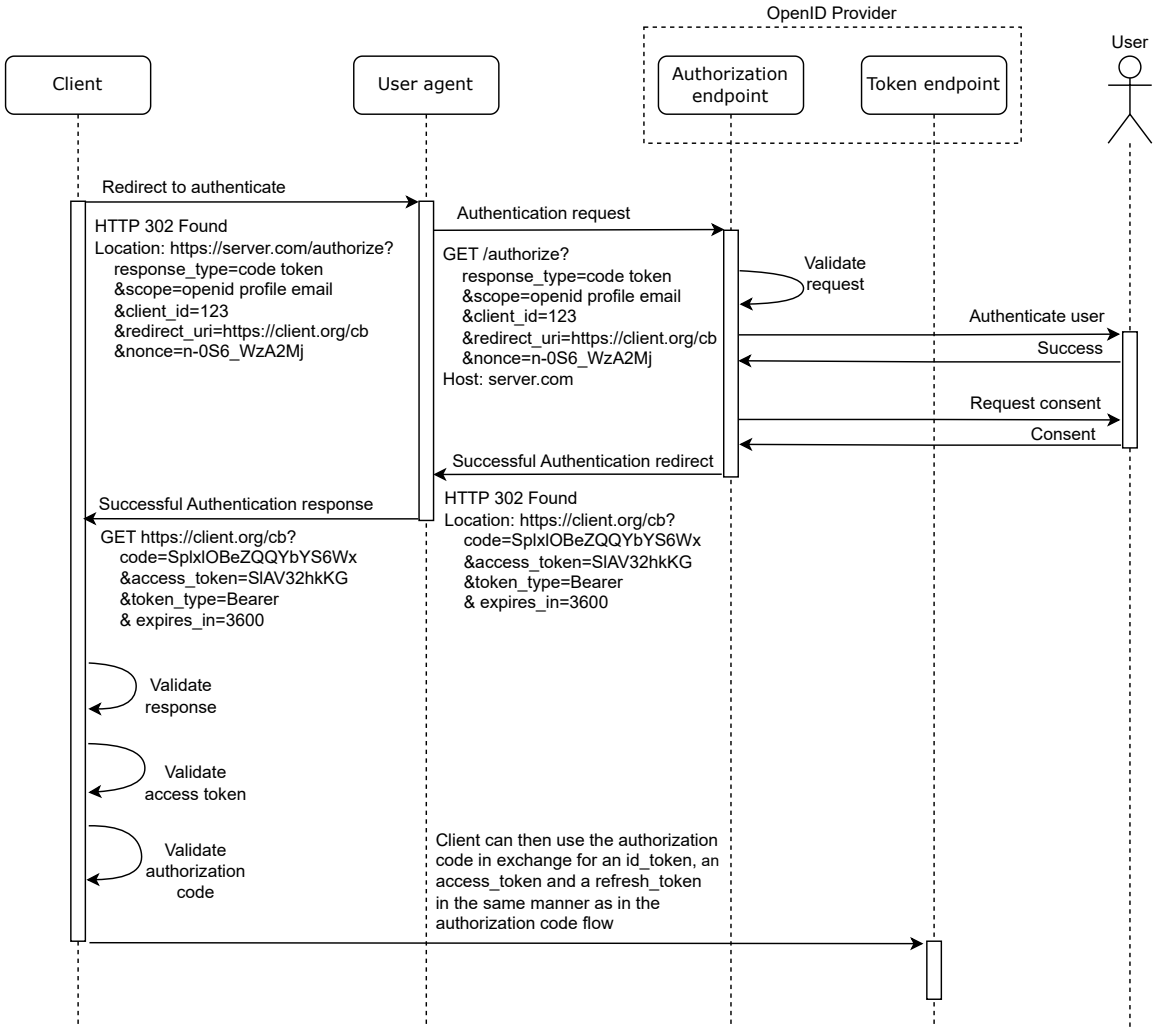


Figure 2.3: Example of a successful **hybrid flow** authentication requesting an authorization code and an access token. The authorization code can be used on the token endpoint precisely as in the authorization code flow pictured in Figure 2.1. HTTP requests and responses are taken from the OIDC specification [18].

Chapter 3

Keycloak Server

Keycloak[1] is an Identity and Access Management (IAM) solution providing centralized authentication and authorization to applications and APIs. It was created in 2014 and had become an upstream project of a Red Hat product – Red Hat Single Sign-On; in 2024, rebranded to Red Hat build of Keycloak[2]. It is written in Jakarta EE, and the Keycloak distribution is based on Quarkus[3]. As of April 2023, Keycloak became an incubating project of Cloud Native Computing Foundation[4].

Keycloak provides an easy solution to securing applications and REST services with minimum effort. This includes storing users, centralized management of users, authenticating, authorizing, and single sign-on for not only web applications. The users can be stored in the local Keycloak user database or federated from an external storage. The storage can be a relational database, Active Directory, or an LDAP server. Social logins are also a viable option. Social Identity Providers can be integrated with a Keycloak instance, enabling delegating authentication to a social media account such as Google, Facebook, Twitter, GitHub, LinkedIn, Microsoft, and Stack Overflow. User accounts can then be managed via the administration console or by users in the account console included in Keycloak.

To secure an application with Keycloak, an administrator must register a client in the Keycloak admin console. Keycloak implements the **realm** mechanism. A realm is a set of users and clients with its own configuration. Keycloak can have a theoretically unlimited number of realms, each completely separate from the others. All applications within one realm „share“ the same user's session, meaning once a user logs in to one of the clients with Keycloak, they are logged in all the applications in this realm. The same applies to logging out. This is the core concept of an SSO[5] application. In a typical scenario, an administrator of an example IT company using Keycloak can have one realm with a database of their employees where they leverage the SSO for their intranet, git repository host, and Google services. Another realm would be for customers and the company customer portal. A mockup of a general communication involving Keycloak is in figure 3.1.

A user can sign into an account in each realm in one moment, but only if these accounts are in different realms. A realm can be considered a singular identity provider. This knowledge is important for the FedCM API[6] implementation.

[1] https://www.keycloak.org//
[2] https://access.redhat.com/products/red-hat-build-of-keycloak
[3] https://quarkus.io/
[4] https://www.cncf.io/
[5] Single Sign-On
[6] Federated Credential Management Application Programming Interface

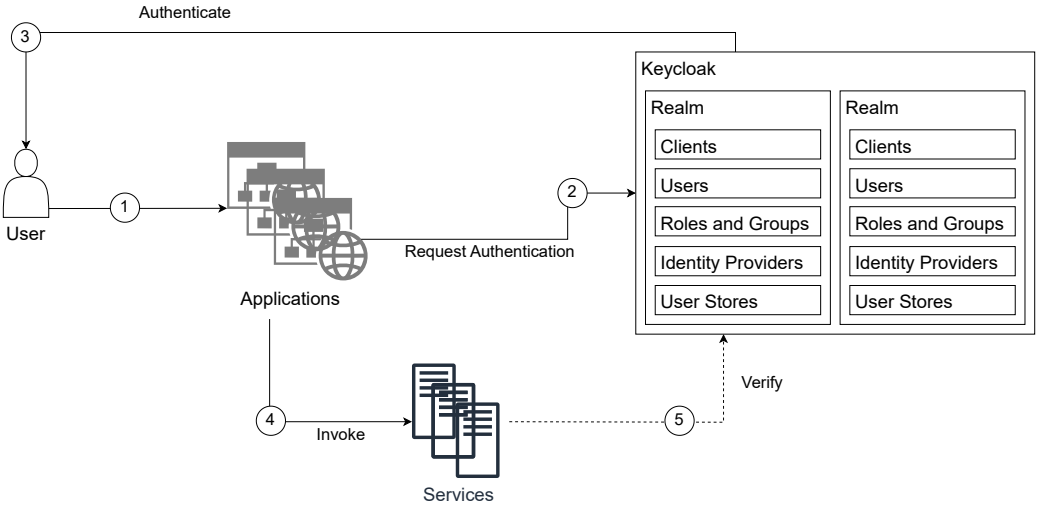


Figure 3.1: When a user accesses an application, it requests authentication from Keycloak, verifying the user's identity and granting access to protected services. This chart is based on a public Keycloak chart [10].

3.1 Javascript Adapter

To help developers with integration with their applications, Keycloak offers several libraries or client adapters in different programming languages and frameworks. These adapters serve different purposes, such as the KeycloakInstalled Java adapter for CLI/Desktop applications, the Keycloak Node.js adapter to protect server-side JavaScript apps, and most importantly, the Keycloak JavaScript adapter. It is a client-side Javascript library called keycloak-js, which operates on the OIDC protocol (discussed more in chapter 2).

This library is used for client-side applications considered public clients, as they can not securely store client credentials. By default, it uses the authorization code flow (see 2.4.1). Still, it can be configured during adapter initialization to use the implicit flow (2.4.2) as well as the hybrid flow (2.4.3). Before its use, the adapter needs to be initialized as shown in figure 3.2.

Once Keycloak is instantiated, the `login` function can be called to authenticate a user for the application. For a better user experience, Keycloak implements OIDC **Session Management** [15] and **Silent authentication**.

3.1.1 Session Management

A session typically starts once the authentication flow is finished and the client validates the ID token. The status of a session is sent as a parameter in the authentication or error response from the Open ID Connect provider. The status of the session can change over time. For example, a user is signed in to two clients, client A, and client B, in one browser window, each in a separate tab. Clients A and B share the same user session. If the user decides to perform a single sign-out on client A, client B must be notified about this change so they can act accordingly.

The specification defines two iframe elements that are embedded in the client. The invisible RP iframe, which is loaded from itself, periodically sends messages to the OP iframe, which checks for changes in the session status. The OP iframe is loaded with an ID as-

```
import Keycloak from 'keycloak-js';

const keycloak = new Keycloak({
    url: 'http://keycloak-server${kc_base_path}',
    realm: 'myrealm',
    clientId: 'myapp'
});

try {
    const authenticated = await keycloak.init({
        onLoad: 'check-sso',
        silentCheckSsoRedirectUri: `${location.origin}/silent-check-sso.html`
    });
    console.log(`User is ${authenticated ? 'authenticated' : 'not authenticated'
    }`);
} catch (error) {
    console.error('Failed to initialize adapter:', error);
}
```

Figure 3.2: An example of initialization of the Keycloak Javascript adapter on a client-side application [11].

signed by the RP and listens for `postMessage` signals from the RP's origin. Upon receiving such a signal, it calculates the session's status, typically using an OP cookie, and compares it to the status passed as a parameter in the `postMessage`. The result is sent back to the RP iframe.

While the concept remains similar, Keycloak is not standard-compliant. It embeds only the OP iframe called the login status iframe. This iframe leverages cookies with a session ID and sends messages to Keycloak to check this cookie. If a change is detected and the user needs to be authenticated again, the client is redirected for authentication to Keycloak, or if configured in the adapter, it can use silent authentication.

3.1.2 Silent authentication

Silent authentication allows a client application to receive tokens from Keycloak without a browser redirecting them to Keycloak. Part of the OIDC core specification describes a `prompt` parameter in the authentication request, which can be set to a value `none`. This tells the authorization server not to display any authentication or consent user interface. The silent authentication, or more suitably re-authentication, creates a hidden Keycloak iframe on the client's side, which makes standard requests to Keycloak. Still, the redirects happen only within this iframe. An existing session with Keycloak identified by an ID in a cookie must exist. Had the session not been there and the user not authenticated, the iframe would have received an error from the Keycloak endpoint.

3.1.3 Browser tracking protection

Cookies are crucial for these two features to work. Unfortunately, they can also be used to track users, so some browsers limit their use or block them completely. Because of this, session management and silent authentication become unfeasible. If Keycloak could still

set these cookies, but the session ID cookie was set for client A, Keycloak could not access this cookie for client B, which is the key for session management. There are mechanisms in place to mitigate this issue, but they spoil the user experience.

In the case of the login status iframe (the session management), the Javascript adapter performs two steps before loading the iframe. In the first step, it checks with the browser's Javascript interface if cookies are enabled, but because the support for this is inconsistent across user agents, it also sets two cookies that serve the purpose of being read in the next step. If, in the call to the second-step Keycloak endpoint, it is found these two cookies from the previous step are not set, the login status iframe would be useless and would not be used. Admins of clients experiencing this issue are advised to set the life span of tokens to a very short time, so in case of a single sign-out, the change in the session status would be found when refreshing tokens. Had the login status iframe been active, the change in session status would have been discovered sooner.

As per the silent authentication and cookies disabled, keycloak-js falls back to the redirect flow. The process stays the same, but the redirect happens on the browser level instead of in an iframe. The user is redirected from a client to Keycloak, and if an active session is found, they are redirected back to the client. The client could be in some state, and after Keycloak redirects back, this state is lost and wiped out.

Chapter 4

Federated Credential Management API

This section paraphrases and describes the proposed Federated Credential Management API solution described in the specification [7], namely the sections affecting the Identity provider. As mentioned, at the time of writing this thesis, it is not yet standardized. It is a community draft report published by the Federated Identity Community Group[1] developed in a public Github FedCM repository[2]. As it is an open specification, multiple articles, blog posts, and GitHub issues are also cited in these paragraphs.

The Federated Credential Management API aims to bridge the gap for the federated identity designs that relied on third-party cookies [7]. The main explainer file for using FedCM states: „The Federated Credential Management API provides a use case specific abstraction for federated identity flows on the web. The identity specific APIs allow the browser to understand the context in which the RP and IDP exchange information, inform the user as to the information and privilege levels being shared and prevent unintended abuse.“ [1]. It is not the goal to replace already used OIDC, SAML, and OAuth but to reuse as much of them as possible.

Similarly to OIDC, it has the Relying Party (**RP**) actor, then an **IdP** (Identity Provider, in the scope of this thesis Keycloak) and a **user agent** (an internet browser). The user agent is placed in the middle as a mediator between the RP and the IdP. The changes impact mostly the user agent and the IdP (in which this API is implemented) and minimally the RP, which makes a simple Javascript call to the user agent to obtain the user's permission to log in with the IdP and receive a token able to access the user's information.

As of writing this thesis, the FedCM is implemented to some extent in Chromium-based browsers, mainly Google Chrome, with plans to implement it in other user agents, such as Mozilla Firefox.

[1] https://www.w3.org/community/fed-id/
[2] https://github.com/fedidcg/FedCM

4.1 Identity Provider HTTP API

The IdP must implement and expose a set of HTTP APIs, which the user-agent calls on behalf of an RP. These endpoints are captured in figure 4.1 and consist of:

- The Well-Known File
- The Config file
- Accounts endpoint
- Client Metadata endpoint
- Identity assertion endpoint
- Disconnect endpoint

All requests sent to the IdP's API must contain the header `Sec-Fetch-Dest` with a newly defined value `webidentity`. This serves as a protection against a Cross-site scripting attack because it cannot be set by random websites [7].

Additionally, all successful requests to the identity provider must respond with a JSON object that can be converted to a respective data structure implemented in a browser.

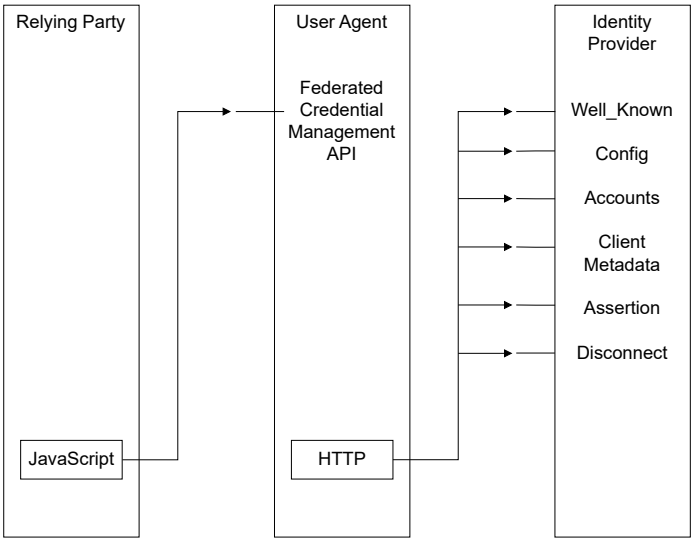


Figure 4.1: An RP notifies a user agent via a JavaScript interface of an intent to log in a user. The user agent then performs a series of HTTP requests to endpoints implemented on the Identity provider's side. This figure is remodeled after a scheme from the FedCM API specification [7].

Each request to the IdP's endpoints contains different information. The data shared to the respective endpoints is shown in table 4.1.

Table 4.1: Information shared in requests to the IdP HTTP APIs. An extended version of a table from the FedCM API specification [7].

Endpoint	Method	Sec-Fetch-Dest header	IdP Cookies	client_id	Origin
The Well-Known File	GET	webidentity	no	no	no
The Config file	GET	webidentity	no	no	no
Account list	GET	webidentity	yes	no	no
Client Metadata	GET	webidentity	no	yes	yes
Identity assertion	POST	webidentity	yes	yes	yes
Disconnect	POST	webidentity	yes	yes	yes

4.1.1 The Well-Known File

This well-known file must be served in a pre-defined location at the root of the IdP. The exact location is `/.well-known/web-identity` of the IdP's eTLD+1, which prevents the manifest fingerprinting attack [7]. This attack can reveal a user's identity without their consent. An eTLD stands for effective top-level domain under which users can directly register their own domains. They are available in a Public Suffix List[3]. The +1 then represents an additional level to the eTLD.

This JSON file contains a `provider_urls` list of URLs pointing at config files. The config files themselves can then be anywhere. At the moment of writing, the number of config files in a well-known file is limited to only one[4].

4.1.2 The config file

This is a manifest file with IdP's HTTP APIs defined in this specification, the name of the IdP, and IdP's branding preferences. The branding preferences modify the prompt a user agent presents to the user when choosing the appropriate account and granting their permission to use this account. They include colors and a displayed icon.

4.1.3 Accounts endpoint

This endpoint returns a list of accounts the user is logged in at the IdP in this browser. Because some IdPs allow more users to be logged in simultaneously, this list can contain multiple accounts. Each account must have an ID (which is IdP-specific) and information about the user. This information is used only to display in a widget, so the user can decide which account they want to use via this GUI[5] element.

For each account, the IdP keeps a list of approved clients. These are the clients for whom this user has already granted permission to use their account. This list is used when displaying privacy policy and terms of service. These policies are described in section 4.1.4. If a `client_id` in this list matches a `client_id` provided by the RP (set up during client registration with the IdP), the policies are not displayed.

Two more lists are returned in the JSON object: `login_hints` and `domain_hints`. An RP can use these to request only accounts matching these hints.

[3] https://publicsuffix.org/
[4] https://github.com/fedidcg/FedCM/issues/333
[5] Graphical User Interface

Because FedCM API does not provide an option to log in directly at the IdP, they must be logged in before initiating the federated login flow.[6]

4.1.4 Client Metadata endpoint

As the name suggests, this interface returns a JSON object with metadata about the client. The specification does not say how the IdP obtains a link for the `privacy_policy_url` and `terms_of_service_url`, which are in the response. When creating the connection between an RP and an IdP account for the first time, these must be displayed to the user. A `client_id` can be utilized by the IdP to decide which client's policies to display.

4.1.5 Identity assertion endpoint

This is the final step in the federated login flow. After the user confirms their desire to connect this account with the RP via a user agent's UI, the user agent sends a POST request to this IdP's endpoint. The request contains a `client_id`, `account_id` of the chosen account, a boolean value whether the user was shown what information will be shared with the RP in the `disclosure_text_shown` parameter, and a `nonce`. If present, the IdP should validate the `nonce` value to prevent CSRF-style attacks. This value was initially set by the RP requesting the user accounts from the IdP; it is encoded in the obtained token and verified to be the same.

After checking the Origin header of the request matches the `client_id`, the IdP returns a token, which is passed back to the client. It is not specified what token should be returned as the FedCM does not rely on authentication or authorization protocols. „The content of the token is opaque to the user agent and can contain anything that the IDP would like to pass to the RP to facilitate the login“ [7]. Upon receiving the token, the RP is expected to validate the token.

With a successful response, the user agent creates a record in its internal storage. Into the **Connected Accounts set** is added a new triple identifying the connection consisting of RP's origin, IdP's origin, and an account ID.

4.1.6 Disconnect endpoint

The disconnect endpoint removes the previously created connection of the account, the RP, and the IdP. It notifies the user agent to remove this record from its connected accounts set. A disconnect request is facilitated in CORS mode with the `client_id` and an `account_hint`. The CORS[7] mode means the request includes an `Origin header`. If the server recognizes the `Origin` of the client, it responds with a header `Access-Control-Allow-Origin` with the originally shared value of `Origin`.

If successful, the IdP responds with a JSON object containing an `account_id`, which allows the user agent to disconnect the account. It also allows the IdP to log the user out of the RP.

[6] There is a proposal for a „button“ mode providing the user agent an option to redirect to the IdP. More described in 4.4.2

[7] Cross-Origin Resource Sharing

4.2 The Browser API

The browser is a trusted party between the user and the identity provider. It exposes an API for an RP and an IdP, which facilitates and intermediates the exchange of the user's identity [7]. The most notable is the **Sign-up/Sign-in API**[8], which allows an RP to initiate the flow via a Javascript call and obtain a token.

This simple call depicted in figure 4.2 passes control to the user agent, which negotiates the federated login, and the client receives a token. As the client receives a token, the user agent creates a new connection with an account, RP's origin, and IdP's origin and stores it in the **Connected Accounts set**. This is a set of triples the user has used FedCM to log in to the RP via the IdP account [7].

Because some calls require IdP's cookies (in the context of a third party), the web browser passes them along as if they were from the same origin.

Part of the browser API is the `IdentityCredential` interface. It is a new type of `Credential`, defined in [19] (currently a work in progress labeled as W3C Editor's draft). This interface exposes a disconnect method which when given `clientId`, `configUrl` and `accountHint` removes the connection account, RP, IdP from the connected accounts set. The use of the disconnect method is in figure 4.3

```
const credential = await navigator.credentials.get({
  identity: {
    providers: [{
      configURL: "https://idp.example/manifest.json",
      clientId: "123",
    }]
  }
});
```

Figure 4.2: A client's Javascript call to initiate federated login via FedCM [7].

```
await IdentityCredential.disconnect({
  configURL: "https://idp.example/manifest.json",
  clientId: "123",
  accountHint: "user@example.com"
});
```

Figure 4.3: A client's Javascript call to log out a user and remove the client-IdP-user connection [5].

4.2.1 The Login Status API

Similarly to the connected accounts set mentioned in the Identity assertion and the Disconnect endpoint, the user agent also holds a **Login Status Map**. This persists data about a connected account of an IdP. The values are IdP's origin and an enum of either `unknown`, `logged-in`, or `logged-out`.

[8] Not to be confused with the Login Status API.

An IdP can modify the set to inform the user agent about the login status via a new HTTP response header `Set-Login`, an example can be seen in figure 4.4. After authenticating a user through its interface, an IdP can then notify the browser about a successful login by including this header in a response[9] with the value `logged-in`.

```
HTTP/1.1 302 Found
Set-Login: "logged-in"
Location: https://client.org/cb#id_token=eyJ0NiJ9.eyJ1c...
```

Figure 4.4: An example of a Set-Login header sent by an IdP after successful authentication in implicit code flow (2.4.2) of only `repsonse_type=id_token`.

An alternative to the HTTP header API is a Javascript API. In that case an IdP can call `navigator.login.setStatus(`*`value`*`)`. Further information about the Login Status API is available in [7].

4.3 Example flow in detail

In an example flow, where a user signs up with FedCM to an RP for the first time (figure 4.5), the user agent's internal Login Status map will be empty. To succeed, the user must be already logged in to an IdP's account. As mentioned in the previous section, the IdP must notify the user agent about the login status via the provided APIs. Once the client decides to obtain the user's federated identity, it calls the browser's JavaScript interface. The user agent takes control and mediates the exchange with IdP via HTTP.

The client must be aware of where the config file is located. The browser makes two requests in parallel, one fetch for the config file and one for the well-known file. Had the config file not been among the URLs in the `provider_urls` list in the well-known file or the list size greater than 1, the user agent would throw an exception, and the flow would have ended. This approach mitigates Manifest Fingerprinting attacks, further reading in section 7.3.1. of [7]. The contents of the config file are URLs for the rest of IdP's endpoints which were until now unknown to the user agent (along with the branding preferences, see 4.1.2.

Another request is sent to the accounts list endpoint with IdP's cookies. The IdP gathers information about all signed-in users in the browser (if it allows more than 1 user) and returns a list in a JSON object of items of type `IdentityProviderAccount`[10]. Because it is the first time this account is used for federated login for this client, the returned account has an empty list of `approved_clients`. One more request is sent in the background to get the URLs for the RP's terms of service and privacy policy. IdP can distinguish the files by a client ID in the request parameters. This implies the IdP is aware of these two items in advance.

This is enough information to present the user with a widget prompt where they choose an account. With a name, an email, and a picture, the dialog displays what information will be shared with the RP and renders 2 links to the policies gained in the previous step. Once this flow finishes this RP is saved in the list of approved clients and there will be no need to show these two links in the widget again, see figure 4.6.

[9] This response can also be an http-redirect.

[10] https://fedidcg.github.io/FedCM/#dictdef-identityprovideraccount

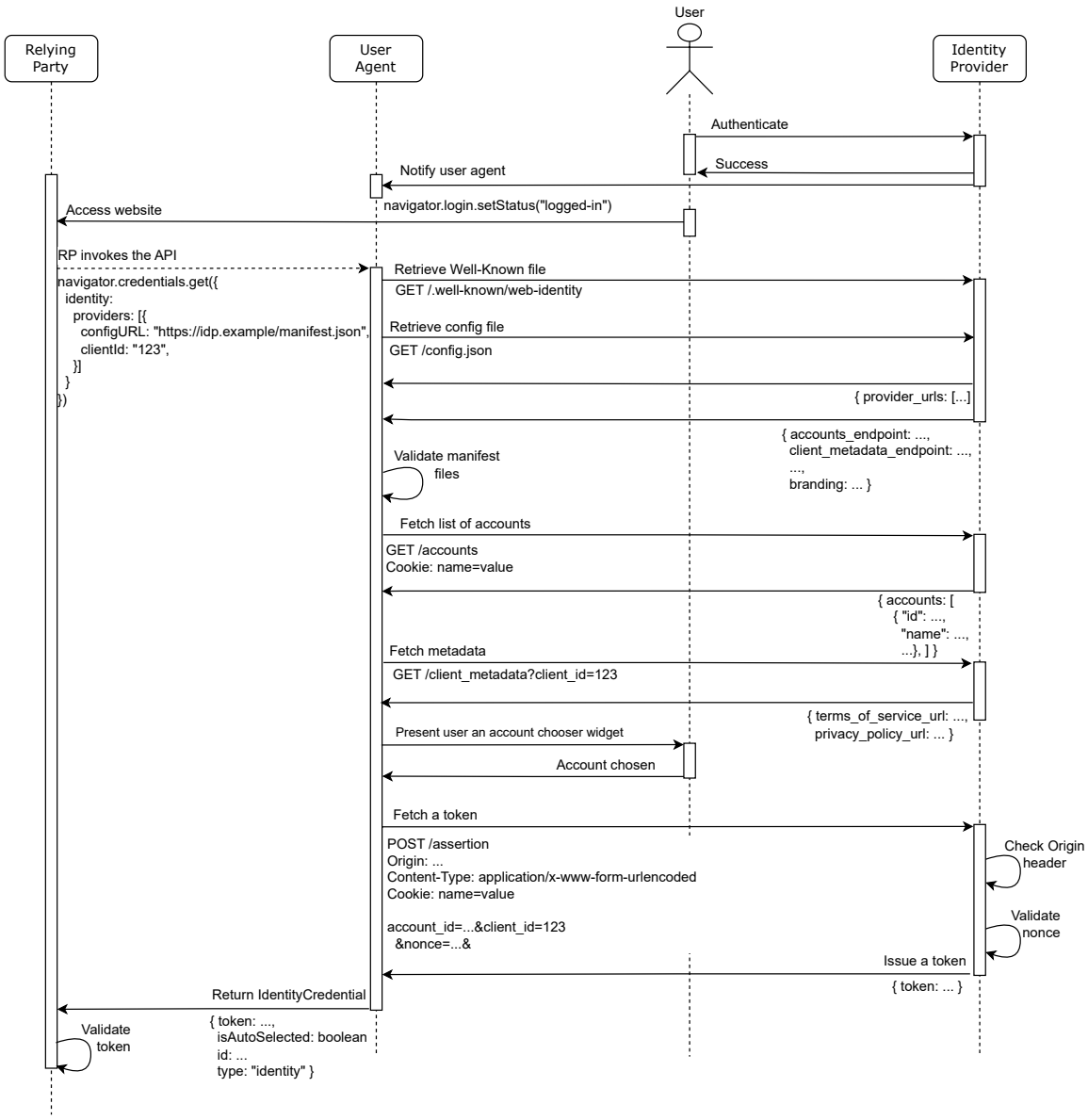


Figure 4.5: A FedCM successful first-time login flow. An extended diagram from the specification [7].

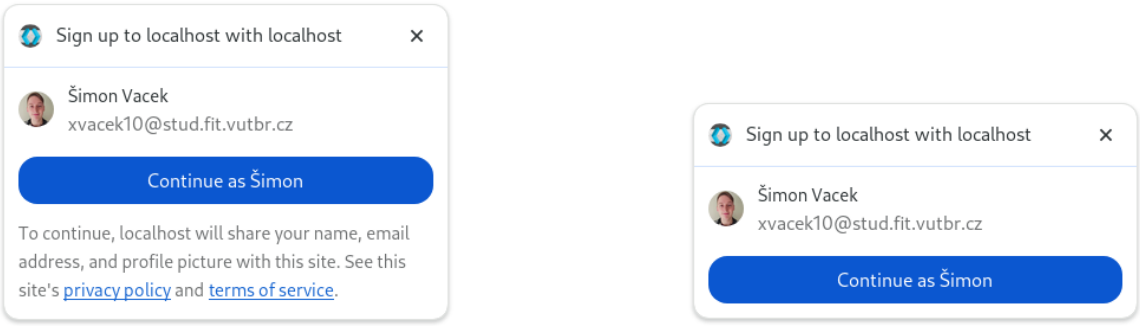


Figure 4.6: Browser presenting an account chooser for a first-time user (left) and a returning user (right).

If the user decides this is what they want and presses a continue button, the user agent creates and sends the last request to the identity assertion endpoint. It is up to the identity provider to ensure the Origin header value corresponds to the client ID so a malicious RP does not receive a token. The client ID is IdP-specific, so the browser can not do this validation. The IdP is also advised to validate the nonce in the parameter. The IdP then issues a new token and sends a JSON object, which must have a single item, a token. While this communication is happening in the background, the user interface is changed to let the user know work is being done (see figure 4.7). User agent processes this and returns the RP an `IdentityCredential`[11] object containing the token.

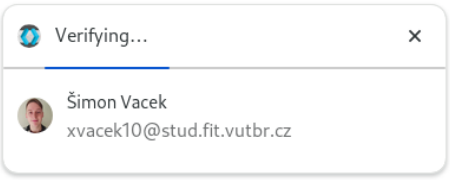


Figure 4.7: The account chooser widget notifies the user the login is being processed.

In a logout flow (figure 4.8), the RP calls the the `IdentityCredential.disconnect` method, and the user agent takes control. Same as in the sign-in/sign-up flow, a browser fetches the manifest files and validates them in the same way. Thereafter, it requests logout on the disconnect URL from the config file. The IdP finds the ID of the account being disconnected and returns it. With this, the user agent removes a connection of (account ID, RP origin, IdP origin) from the Connected Accounts set.

[11] https://fedidcg.github.io/FedCM/#browser-api-identity-credential-interface

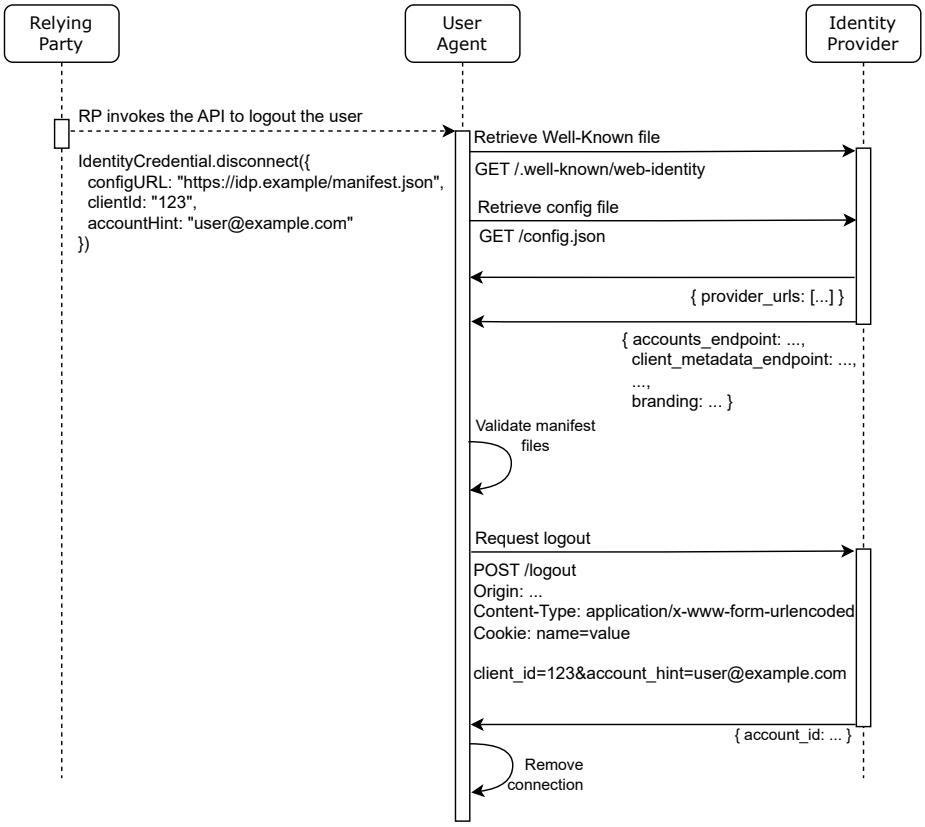


Figure 4.8: A successful FedCM sign-out flow.

4.4 Proposals and extensions

While the FedCM API specification is a draft and could be labeled as unofficial, there are also proposals and experimental features that are not part of the specifications. These features are implemented exclusively in the Google Chrome browser. Some of these features are hidden behind a feature flag which enables them.[12]

4.4.1 Auto-reauthentication

Auto-reauthentication is designed to improve the user experience for those who „used only one account with FedCM API to sign into the website on this browser and the user is signed into the IdP with that account“ [4]. It offers a faster way to proceed without user interaction; a widget showing the login process is still displayed but does not require the user to click the „Continue“ button, see figure 4.9.

The RP can negotiate auto-reauthentication with a `mediation` parameter. The following code snippet demonstrates its use:

[12]The flags can be turned on inside Google Chrome when accessing `chrome://flags/` and searching for „FedCM“

```
const cred = await navigator.credentials.get({
  identity: {
    providers: [{
      configURL: "https://idp.example/fedcm.json",
      clientId: "1234",
    }],
  },
  mediation: 'optional', // this is the default
});
```

In the context of FedCM, it can be of four values:

- `required`: Auto-reauthentication is not possible, and the user must always click the „Continue“ button
- `optional`: Use auto-reauthentication if possible; fall back to the user mediation otherwise
- `silent`: Use auto-reauthentication if possible; fail the flow silently if not

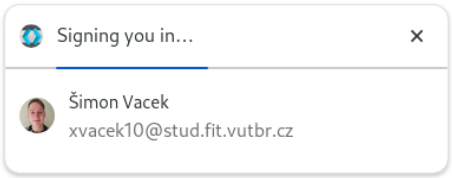


Figure 4.9: A user agent's slightly different widget showing a user is being automatically reauthenticated.

4.4.2 Button mode

This proposal suggests having an alternate flow, a „button mode,“ which could be chosen instead of the current „widget flow“. Unlike the widget flow, the button flow is not meant to be invoked when the user lands on RP. Instead, it is used after the user performs an action, like clicking a button.

The motivation for it is the current inability of a user to log in with their account if they are not already logged in at the IdP. In the current widget flow, if the user is not signed in to their account, the accounts list endpoint 4.1.3 returns an empty list, causing the federated login to „fail silently,“ and the user does not receive any message about this. The idea is, in the bigger widget (displayed on figure 4.10) of the button flow, the user is instead presented with an option to sign in to the IdP in a pop-up window and, after successful authentication, continue in the fedcm flow.

This was discussed and summarized in a comment on „A not-yet logged in IDP has no route to success with this flow“ issue[13], which led to „An intent to Experiment: FedCM Button Mode API and Use Other Account API“[14]. This feature can be turned on with flag `#fedcm-button-mode`.

[13] https://github.com/fedidcg/FedCM/issues/442#issuecomment-1949323416
[14] https://groups.google.com/a/chromium.org/g/blink-dev/c/bQqXXv2S9q0/m/XDBDuhr0AgAJ

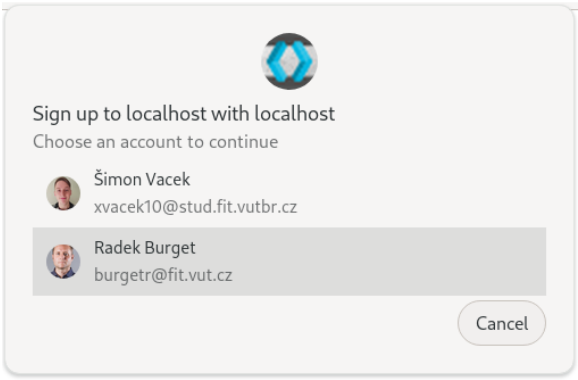


Figure 4.10: An account chooser in button mode is a slightly larger pop-up, here displayed with two users who have this client in a list of approved clients.

4.4.3 Dynamic sign-in flow

This similar feature enables users to sign in to an IdP account. In this case, however, the prompt to sign in on the Identity Provider is presented only if the user was signed in, but the session expired, or the sign-in status was not updated. It means the browser expects a user to be signed in with the IdP (based on the record in the Login Status Map), but the IdP's accounts list endpoint returns an empty list [2].

The IdP provides a `signin_url` in its config file for the dynamic sign-in flow to work. After the user authenticates in the pop-up window, the IdP notifies the user agent about a successful login described in section 4.2.1 and calls `IdentityProvider.close()` to close the popup window. See figure 4.11 for the implementation in Google Chrome. This feature is enabled with `#fedcm-idp-signin-status-api` flag.

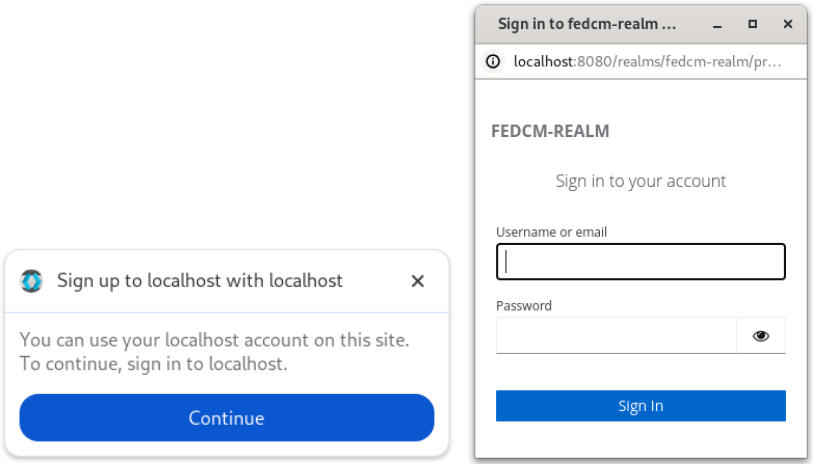


Figure 4.11: When a user is supposed to be signed in but is not, a different widget opens up (left) asking them to sign in with IdP, which opens a new window (right) for authentication)

Chapter 5

Design

The motivation for implementing and integrating the FedCM API into Keycloak is the third-party cookie phase-out. Keycloak could not provide users the same service and experience without their usage. The problem described and the solution to the phasing out (see section 3.1.3) are unsustainable. With FedCM API, at least a session negotiation for a client application could happen. It could answer the following questions:

- Is a user authenticated?
- Who is the user?
- Which account does the client and user want?
- Is there an active session? Can it be given to me?

Ideally, the whole Keycloak Javascript adapter could be removed. Further experimenting with FedCM and future additions to the specification will clarify whether it can be a complete replacement. The current specification avoids being tied down with existing protocols and focuses on identity, not session management. The goal of this thesis, in particular, is to provide a proof of concept and decide if the FedCM API is a good match for Keycloak.

5.1 Keycloak Service Provider Interface

The most suitable option for implementing and integrating the FedCM API into Keycloak is to use a Service Provider Interface (SPI). This design pattern is used in Java ecosystems to enable better extendibility of existing applications, even for third parties, without modifying the original codebase. The SPI does not need to be compiled and packaged with the original application. It can be delivered later and loaded dynamically at runtime [17]. Keycloak uses the SPI for many components, allowing the community to make custom extensions such as Keycloakify[1] for creating Keycloak themes with React or Python client[2] library for Python applications.

A custom SPI can be created, or an existing one can be overridden. An SPI fundamentally defines a contract for certain functionality within Keycloak. Keycloak has two key interfaces: `Provider` and `ProviderFactory`. An SPI must implement these two interfaces

[1] https://www.keycloakify.dev/
[2] https://github.com/keycloak-client/keycloak-client/

(or create interfaces extending them and have classes implementing those). Additionally, a service configuration file [13] is needed.

A provider factory creates instances of a given provider that has the actual business logic. There is only one instance of a factory per Keycloak server. This factory must be registered in a configuration file in the `resources/META-INF/services` directory. Figure 5.1 depicts an exemplary directory hierarchy of an SPI.

The semantics are that the name of the configuration file corresponds to the full name of the factory (including the package) this factory is implementing and contains an item with this factory's fully qualified class name. This is used at Keycloak boot time when a Provider Loader scans for and loads all provider factories.

For example, in this implementation, there is a `FedCMProviderFactory` in package `org.keycloak.fedcm` that creates instances of `FedCMProvider`. It implements the `Realm ResourceProviderFactory` interface, so the service configuration file would look as follows:

org.keycloak.services.resource.RealmResourceProviderFactory

```
org.keycloak.fedcm.FedCMProviderFactory
```

```
src
└── main
    ├── java
    │   └── org
    │       └── keycloak
    │           └── fedcm
    │               ├── FedCMProviderFactory.java
    │               └── FedCMProvider.java
    └── resources
        ├── application.properties
        └── META-INF
            ├── beans.xml
            └── services
                └── org.keycloak.services.resource.RealmResourceProviderFactory
```

Figure 5.1: Example of a Service Provider Interface directory hierarchy compliant with Keycloak the SPI.

The concept of providers is widely used across Keycloak. Thanks to that, many components can be modified to change the behavior. For the `Provider` interface, there are almost 700 implementations of it in Keycloak. One example of this design pattern is the User Storage SPI. Keycloak offers integration with various storage options for users. One may still find no suitable option for their needs. If so, they can extend Keycloak to support their user storage. A developer can implement their version for the `UserStorageProvider` and `UserStorageProviderFactory`. The methods defining, locating, and managing users are in other capability interfaces from which a developer can choose. These include `UserLookup Provider` (for methods needed to log in users from the custom storage), `UserQueryMethods Provider` (for more complex user queries), or `UserRegistrationProvider` (to support modifying the storage – add and remove users). [13]

Let's say a developer wants to extend Keycloak to support CSV files as storage. They would have a class `CsvUserStorageProviderFactory` and `CsvUserStorageProvider`. Their relation to the base user storage interfaces is illustrated in figure 5.2. In the provider class, the developer can implement methods like `getUserById(userID)` to iterate through the file lines to find the user. Once the user is found, it must be mapped into an implementation of `UserModel` and returned. For some methods from the user storage SPI (`getUserById` in particular), all storage providers implementing these interfaces are looped through until one implementation does not return null. Other configurations are not necessary. Keycloak automatically finds the storage provider if the required interfaces are implemented and registered in a configuration file.

A developer is not required to wire it further into the codebase. These SPIs are allowed to be packaged standalone. Assuming the CSV user storage provider is a maven project, only a dependency for artifact `keycloak-server-spi` is added to the pom file. To register the whole provider, the project will be built and packaged in a JAR[3]. The developer downloads a Keycloak distribution from the official website[4] and extracts the Keycloak archive. The JAR archive with a custom SPI is placed in the `keycloak/providers` directory. Keycloak can then be built with `./keycloak/kc.sh build` and afterwards run with `./keycloak/kc.sh start` (if the server is not running in production but only in development mode, `start-dev` can be used instead). The user storage SPI is loaded automatically.

The following section defines the providers and their factories used in the implementation.

5.2 The Federated Credential Management API design

The Service Provider Interface can be leveraged to add custom REST endpoints to the Keycloak server. Based on the specification, the endpoints can be divided into two groups. As mentioned in section 3, each realm within Keycloak has its own set of users and clients. Each realm is a separate identity provider. One group of endpoints is realm-based, which includes all of them except the well-known file endpoint. The second group is only the well-known file that needs to be globally accessible at the root of the Keycloak server.

Based on this, a design for two providers can be proposed. The first is the `FedCMProvider` and its factory. The second needs to be accessible at the root, and Keycloak does not have an SPI hooking REST endpoints at the very root. Two ways can be taken. One would be a `WellKnownFileResource` class with jax-rs annotations hooked directly to the `/.well-known/web-identity` path. But with best practices in mind, a more suited solution is to define a reusable `RootResourceSpi` interface serving endpoints at the root along with interfaces `RootResourceProvider` and `RootResourceProviderFactory`. Our well-known file endpoint can be served in new classes implementing these interfaces.

As per the `FedCMProvider`, thanks to implementing the `RealmResourceProvider`, it will have access to additional information needed for a FedCM flow, such as the user model, the client model, and authentication services.

This design allows the whole implementation to be a standalone extension of the Keycloak server, which can be added as described in section 5.1

See figure 5.3 for a complete overview of the designed classes and interfaces that implement the necessary SPIs.

[3] Java archive

[4] https://www.keycloak.org/downloads

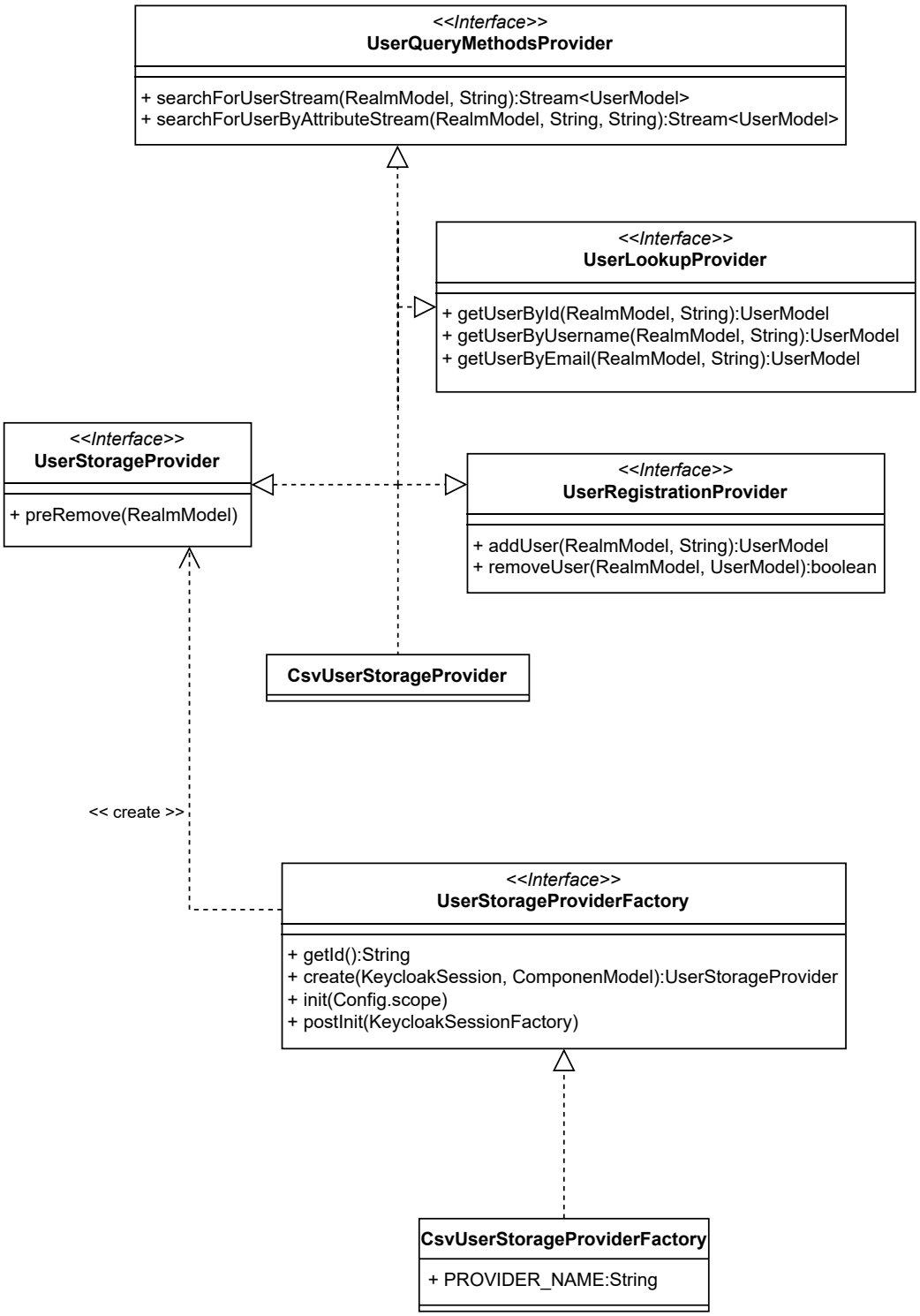


Figure 5.2: A class diagram with a custom CSV user storage SPI example. Note that the entities do not include all fields and methods from the real interfaces, as they add no extra value to this demonstration.

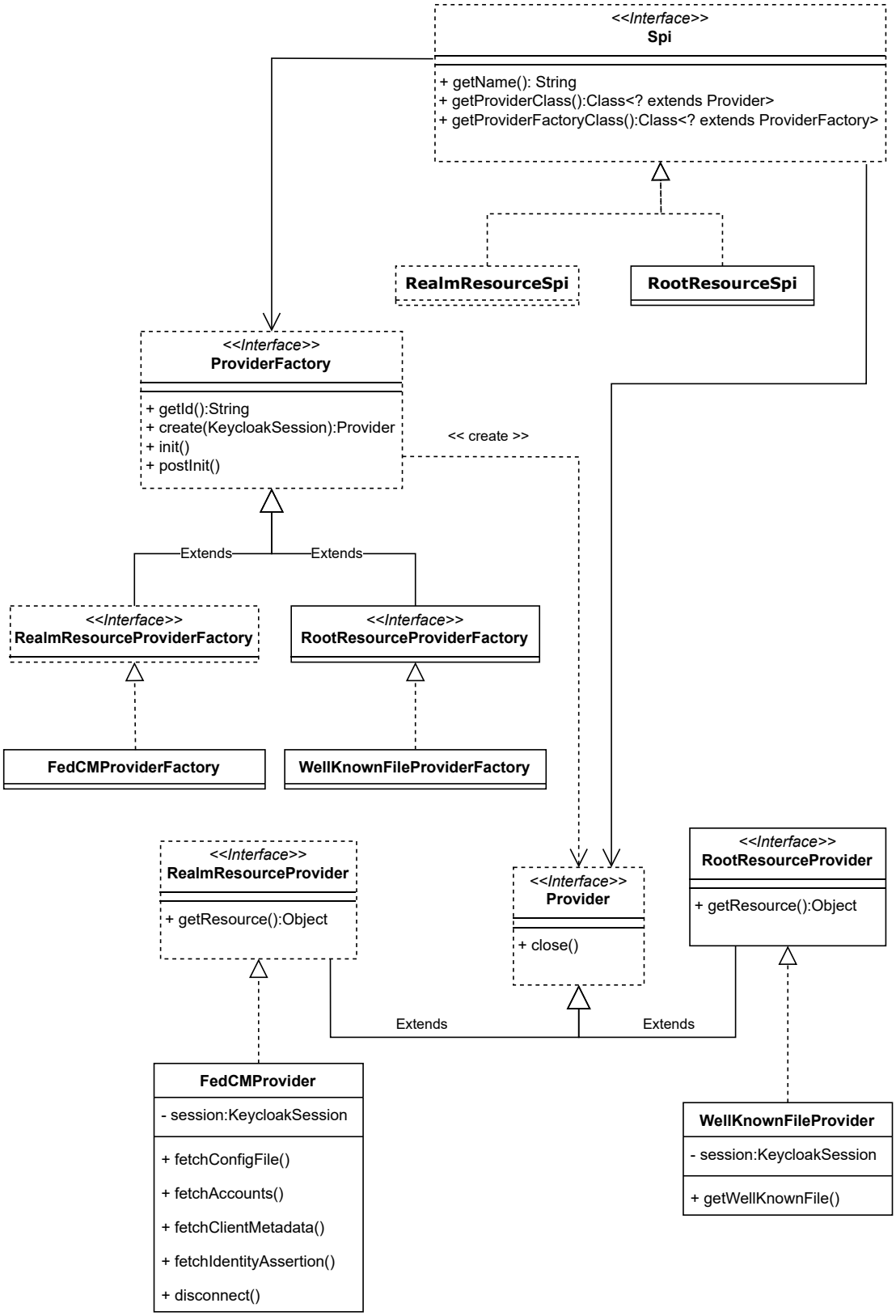


Figure 5.3: A class diagram of interfaces and classes designed for the FedCM API integration. Entities with dashed frames denote interfaces already implemented in Keycloak.

Chapter 6

Implementation

Implementing the FedCM API specification was problematic at times. As it is a new technology, a developer must follow the specifications and explore other sources. Currently, FedCM is implemented in the Chrome browser in the Origin trial status. The specification is open for feedback, so it can change to fulfill the needs of all parties involved. A prototype is expected to be built to find what is missing and provide feedback to the community group. Therefore, some features are implemented in the origin trial before adding them to the specification (also mentioned in section 4.4). The most resourceful proved to be the Privacy Sandbox blog[1] and issues tracked on the GitHub repository[2]. This could also mean some exciting and crucial features can be simply overlooked. Let's start with implementation details before moving on to proposals for the future.

6.1 Endpoints Implementation

The design outlined in figure 5.3 applies. The provider classes implement all the necessary endpoints. These endpoints use the jax-rs annotations from the package `jakarta.ws.rs` to define the HTTP method, path, request parameters, and other relevant configurations. The user information used in the `fetchAccountsList()` method and the following implementations of endpoints are thanks to the `KEYCLOAK_IDENTITY` cookie passed in the request. As shown in figure 6.1, if a user has signed in with Keycloak, an instance of `AuthenticationManager` can be used to obtain an `AuthResult` in a given realm containing authentication-related information, including the `UserModel`.

```
AuthResult authResult = (new AuthenticationManager())
    .authenticateIdentityCookie(session, realmModel);
if (authResult == null) { // user is probably not authenticated
    throw new WebApplicationException();
}
UserModel user = authResult.getUser();
```

Figure 6.1: Example of obtaining a `UserModel` of an authenticated user in Keycloak.

[1] https://developers.google.com/privacy-sandbox/blog/
[2] https://github.com/fedidcg/FedCM/issues

Information that needs to be stored and remembered is harder to implement in Keycloak. The specification lists two things: the list of approved clients returned from the accounts endpoint 4.1.3 and the client metadata in the client metadata endpoint 4.1.4. For each user, there is a collection of attributes. They typically hold user information such as name, email, and other affiliated metadata [12]. This collection allows the mapping of multiple values to a single key, which is why it was used for storing clients consented to by a user.

Some of these attributes can be managed on the user profile. A user profile attribute can be added and configured in the admin console and then optionally managed by users in the account console. As profile pictures are not in Keycloak by default, they can be added this way. When retrieving a picture in the accounts endpoint, if there is no value set for this attribute, the JSON field is simply filled with an empty String, a valid response to the FedCM API.

The privacy policy URL and terms of service also don't cause an error if not supplied but should be if available. My approach is sufficient for a prototype but needs future improvement. It makes use of client policies. These are intended to dynamically enforce security configurations and compliance requirements across different clients based on predefined conditions. Clients also have a collection of attributes; however, unlike the user attributes, they can not be accessed and managed in the admin console. A Keycloak administrator is expected to create a role „policies“ and provide links for two attributes: privacy-policy and terms-of-service. The names must perfectly match, and their definition is in figure 6.2.

```
"roles": {
    "client": {
        "example-client": {
            "name": "policies"
            "attributes": {
                "privacy-policy": ["example-client.com/privacy-policy/"]
                "terms-of-service": ["example-client.com/tos/"]
            }
        }
    }
...
}
```

Figure 6.2: A pseudo JSON object showing the expected role definition and values for a model example-client.

There is currently no proper solution for the branding preferences of Keycloak supplied in the config file (see 4.1.2). To simply demonstrate their use, they are fixedly hard-coded in the implementation.

6.1.1 Connecting a user account

Keycloak, as an Identity Provider, must notify the browser when a user logs in. This is the only instance when the original Keycloak codebase has been tampered with. After a successful login, Keycloak returns a redirect response of type `jakarta.ws.rs.core.Response`. This response is extended by the extra header `Set-Login: "logged-in"` (figure 4.4).

When the identity assertion endpoint receives a request, Keycloak mints a token and attaches a client session to the user session. Before that, an authentication session model is created and supplied with authenticated user and client scopes. These scopes are crucial for the token as it needs them to determine what information it authorizes access to. The nonce passed as a request parameter will be included in the token and can be checked by the client, who is waiting for a response from the user agent. With the specification being vague about the token classification and only one requirement, which is that the response from this endpoint must return a JSON object with a single value, three credentials could be minted. An ID token, an access token, or an authorization code. Knowing the client desires a credential to access the user's resources, the access token is the only choice for Keycloak.

With sessions prepared, a `TokenManager` generates a new access token. At this point, the user has granted consent to use their account at the RP and should be put on the list of approved clients.

This endpoint also uses the proposed Error API. It allows Keycloak to signal errors encountered during identity assertion. Through a JSON object, Keycloak declares a type of error and a link to a page with information about the error [3]. It is realized with an `enum` structure `ErrorTypes` with values being directly convertible to a jax-rs Response type. In this prototype iteration, the links lead to a custom error endpoint that redirects to online Keycloak documentation, with a link to a relevant chapter, see figure 6.3.

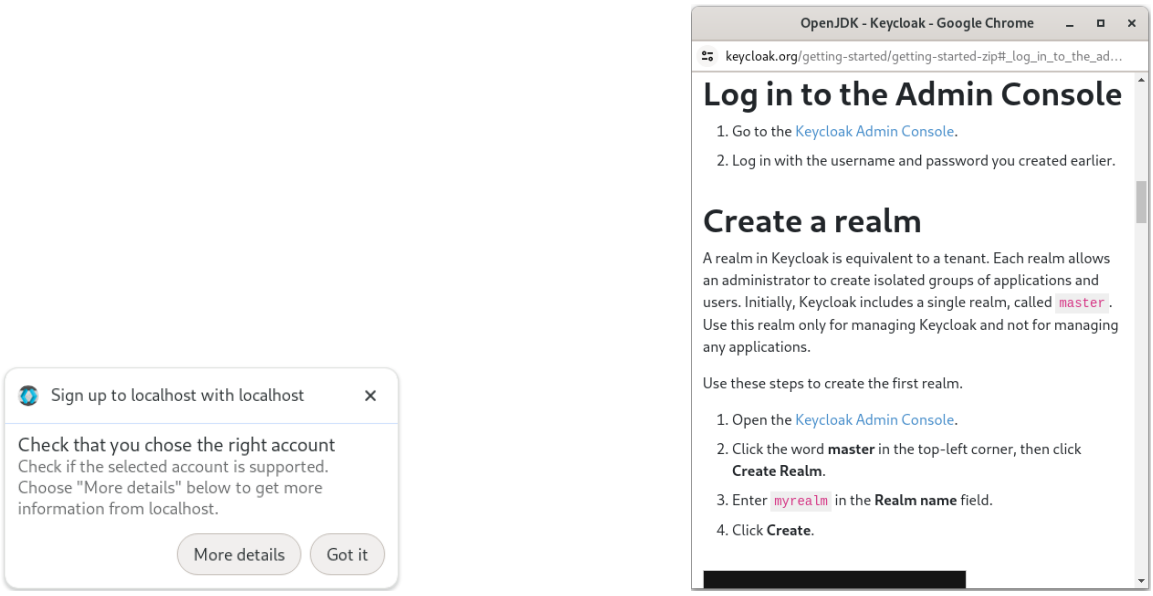


Figure 6.3: A browser UI shown to a user when an error during identity assertion is encountered - utilization of Error API.

6.1.2 Disconnect and logging-out

The disconnect endpoint is responsible for disconnecting a previously made federated login connection [7]. It is unclear what exactly happens on IdP's side. Keycloak could sign the user out of all applications, terminating their session. Or terminating only the client session and leaving the user signed in to the rest of the clients. This uncertainty was

brought up on FedCM GitHub issue[3]. Only the client session was decided to be terminated by modifying the set of client sessions associated with the user. The user can log in with FedCM again and possibly sign in with another identity provider. If the user pleases, they can sign out of all clients, for example, in the account console. Keycloak notifies the browser about a login status change in the same manner as when a user signs in during a regular OIDC flow.

6.2 Implementation plans in the future

The current prototype certainly has flaws that need fixing. The obvious is the branding preferences dictionary, which could be integrated with customizable Keycloak themes. Keycloak server administrators could modify the visuals of the visuals based on their deployment and preferences. Another issue is the dynamic sign-in flow (see section 4.4.3). While it is possible to engage in the flow, and a pop-up window appears where a user can sign in, Keycloak is expected to close this window with `IdentityProvider.close()` once the authentication is successful. This does not happen now. This process could be automated by including a parameter, for example, `fedcm_popup=true` in the `login_url`. This parameter would be passed along in the redirection URI to the OIDC authorization endpoint. Once a user signs in, Keycloak can propagate the value again to the account console, which is redirected to. If this value is detected, the Javascript could execute closing the pop-up window.

6.2.1 The Well-Known File location

The Well-Known file needs to be served by an IdP on an eTLD+1 on path `/.well-known/webidentity`. That is a problem for these reasons:

- Keycloak can not guarantee where a customer or system administrator deploys the Keycloak server. The most common deployment is `keycloak.mycompany.com`, where `mycompany.com` would be an eTLD registered in the public suffix list (4.1.1). However, the structure is entirely up to the Keycloak server administrator.
- Each realm is essentially an identity provider with its set of FedCM endpoints. The config file has to be served in each realm, so the well-known file has a list of `provider_urls` containing all the realms. The specification currently does not allow to have more than one config file. That is not a big issue. Because of security reasons, Keycloak does not want to expose all available realms publicly.

A reverse proxy could be used on the Keycloak side to mitigate these issues. A well-known file will be served in each realm, so only one config file is present in `providers_urls`. Keycloak will provide documentation for server administrators on how to set up the proxy, which would forward requests to Keycloak and appear to the client as if the well-known file was on the root. The proxy would do as follows:

$realm\text{-}name$-keycloak.mycompany.com/$.well-known/web-identity$
->
keycloak.mycompany.com/$realms/realm\text{-}name/fedcm/.well-known/web-identity$

[3] https://github.com/fedidcg/FedCM/issues/558

6.2.2 Merge into the upstream

The proof of concept is packaged as an SPI or an extension. It is not part of the main codebase. Users and clients wanting to try it out are free to do so. In the near future, the implementation should become part of Keycloak. The FedCM API would be a feature not enabled by default but turned on by a parameter at Keycloak start-up. This way, Keycloak users can try it with their applications and provide additional feedback that benefits its development. The feature would be purely experimental and not advertised as a supported solution. At least not for now.

The feature could be enabled per realm in the Keycloak admin console. A UI change enabling a FedCM feature had already been made at the beginning of implementation (displayed in figure 6.4) but was abandoned because it required further changes to support and was not the main goal of this thesis.

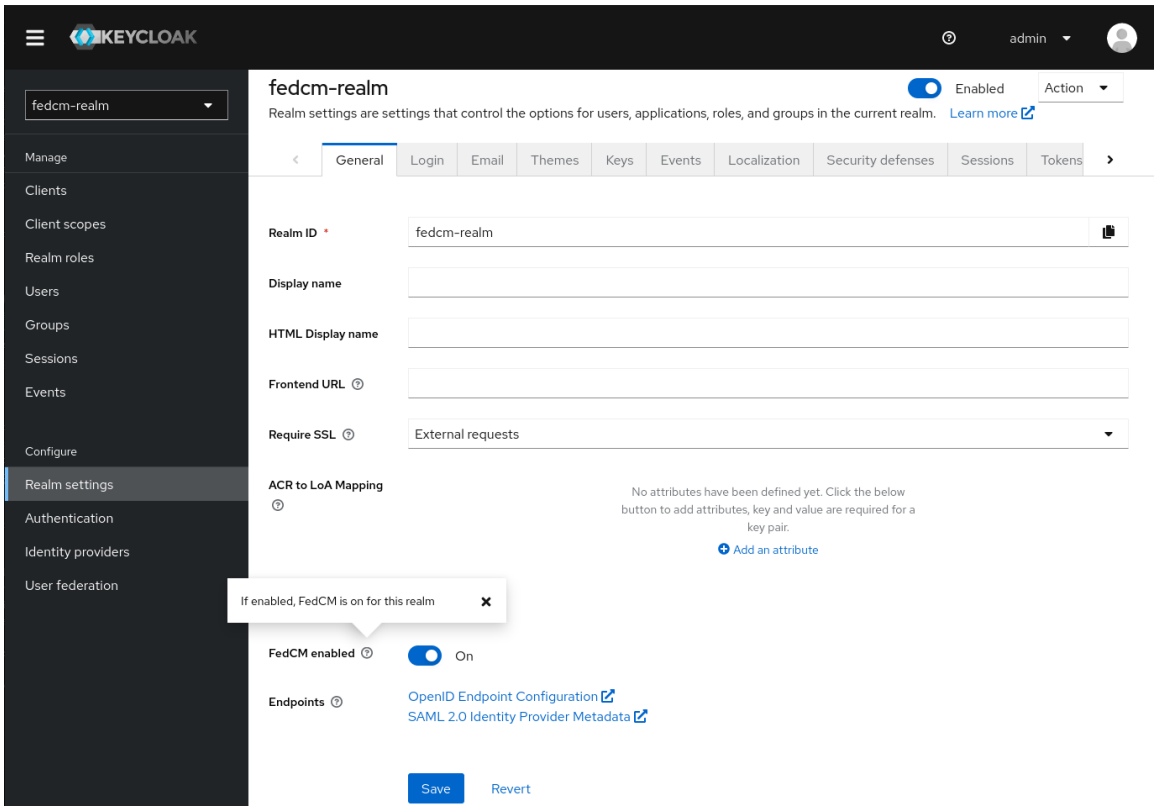


Figure 6.4: A feature toggle available to Keycloak administrators enabling the FedCM flow within a realm.

6.3 Authorization extension

A crucial extension of the Federated Credential Management API was discovered very late in development. Until now, how FedCM deals with authorization and how clients can request OAuth 2.0 scopes to access different Keycloak functionality has not been discussed. When the OAuth 2.0 protocol was built, certain measures had to be taken to make it more secure. One of the key elements was transparency to the resource owner and keeping them informed at all times. The user should always be in control of the authorization process

and get all information to make informed decisions. This is realized by presenting a user with a user consent form [14]. This is specifically used and required for public clients without a secret. These are the clients secured by the Keycloak Javascript adapter. The Federated Credential Management can obtain the user's consent (shown in the left picture of figure 4.6). Still, it is only for the basic scope represented in Keycloak as a „profile."

The authorization extension for FedCM enables the client to request additional non-profile OAuth scopes and other parameters relevant to the IdP [6]. It also introduces a new field list with parameters that are irrelevant to the user agent but are to the identity provider. When the browser detects non-profile scopes, it does not recognize them and can not prompt the user to consent to them. Simply because the browser does not know the semantics, it does not know what, for example, „calendar" means. So, the FedCM flow is modified so that a new IdP window pops up where the consent is obtained, and the flow returns.

A Javascript call requesting other scopes would be modified this way:

```
const credential = await navigator.credentials.get({
  identity: {
    providers: [{
      configURL: "https://idp.example/manifest.json",
      clientId: "123",
      responseType: ["id_token"],
      scope: [ "profile", "email", "phone", "roles" ],
      params: {
          "code_challenge": "...",
          "code_challenge_method": "plain"
      }
    }]
  }
});
```

Modifications to the flow occur during the identity assertion. After the user chooses an account, the user agent looks at the list of scopes. If the browser finds non-profile scopes, it sends a request to the IdP with the scopes in parameters and allows it to return a JSON object with a `continue_on` URL instead of a token. This is an address on IdP's domain where it can display a consent form listing the shared information with the RP in its own words. It happens in a pop-up window. After the consent is gathered and a token is minted, the IdP can close the window and send the token with the identity provider interface: `IdentityProvider.resolve(token)` and `window.close()`.

This addition to the specification is the missing piece Keycloak very much needs. It also resolves the question about the token, but sadly, it is not currently implemented. With so many sources for the implementation, it is not easy to find all the information. Missing this feature in the implementation is simply an error by the author. Still, this extension is not part of the specification and, therefore, not in the scope of this thesis.

Chapter 7

Testing

The implementation of the FedCM API on the side of Keycloak is not very graphic. The API does nothing until it is invoked by a relying party, a client application. For this reason, a simple Javascript application was also developed to call the browser interface and test the functionality. This application is part of the submission and is described in the appendices. It runs in the browser, in this case, Google Chrome version 124. The whole functionality can be narrowed down to two Javascript calls. One is for the sign-in (see figure 4.2), and one is for the disconnect (or sign-out) in figure 4.3. Both these actions can also be checked in the Keycloak administration console.

Once a user consents to the federated login, the FedCM Keycloak extension creates a session for the client. The session can be confirmed and viewed in the „Sessions“ section of the admin console (figure 7.1) and in the „Applications“ section of the account console (figure 7.2).

For the sign-in, it is vital to verify that only an authenticated user with Keycloak can succeed in the FedCM sign-in flow (unless the button mode is used). If there are no users, Chrome already knows this, as there is no connection in the connected accounts set, and an error is returned. This error message is only displayed in the browser's console and can not be caught and displayed to the user. This is intended as the browser keeps the connected users a secret until they specifically allow it. It returns an „error retrieving a token“ message, the same as when the user cancels the dialog. That is also why the example RP application displays generic error messages.

For testing, Chrome needed to be configured to support other FedCM functionality and ensure, for example, the environment is ready for the third-party cookies phaseout. These flags are:

- `#test-third-party-cookie-phaseout`: Enabled
- `#fedcm-button-mode`: Enabled
- `#fedcm-disconnect`: Enabled
- `#fedcm-error`: Enabled
- `#fedcm-idp-signin-status-api`: Enabled
- `#fedcm-skip-well-known-for-same-site`: Disabled
- `#fedcm-without-well-known-enforcement`: Disabled

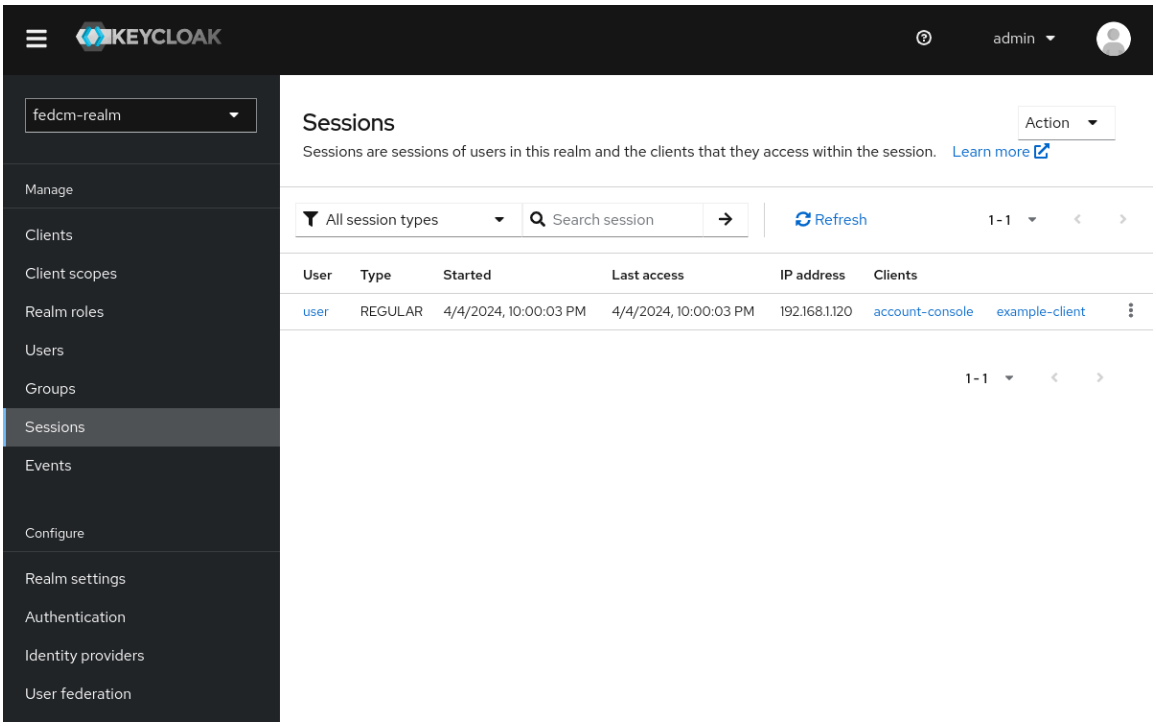


Figure 7.1: Keycloak admin console confirming the user has an active session in example-client.

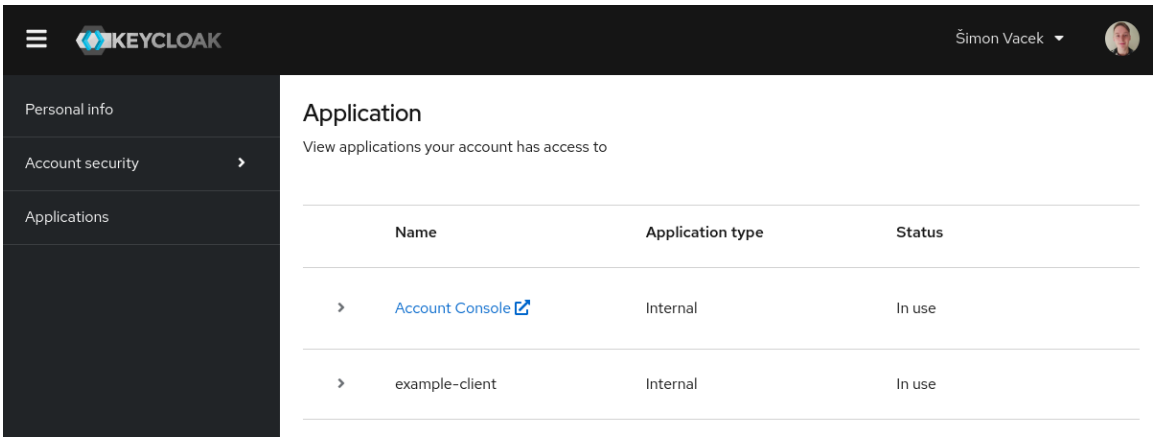


Figure 7.2: Keycloak account console confirming the user has an active session in example-client.

All screenshots included in chapter 4: Federated Credential Management API were taken during testing on the client application. The requirement for the Sec-Fetch-Dest: webidentity in the requests can be tested by a simple comparison of accessing the Keycloak URLs (for example, the well-known file on `keycloak:8080/.well-known/web-identity`) from a browser, which fails with the 400 status code, and via a curl command providing the header:

```
curl -i http://localhost:8080/.well-known/web-identity \
    -H "Sec-Fetch-Dest: webidentity"
```

The rest of the functionality was tested manually in the browser environment. Instructions for verification of the functionality are included on the application's main page, as well as in the appendices. These include checking the token has the user data, ensuring the correct user account is obtained, the client metadata is displayed during the user's first login, the login status is updated, the dynamic sign-in is fired when the session expires, and a scenario where a client with an unknown client ID tries to obtain user's identity.

In figure 7.3 is the client application displaying the user's information after a successful federated log-in. It includes a button to display the JWT access token shown in figure 7.4.

During this process, Chrome sometimes disables third-party sign-in. When that happens, it needs to be re-enabled for the client in the settings.[1]

[1] `chrome://settings/content/federatedIdentityApi`

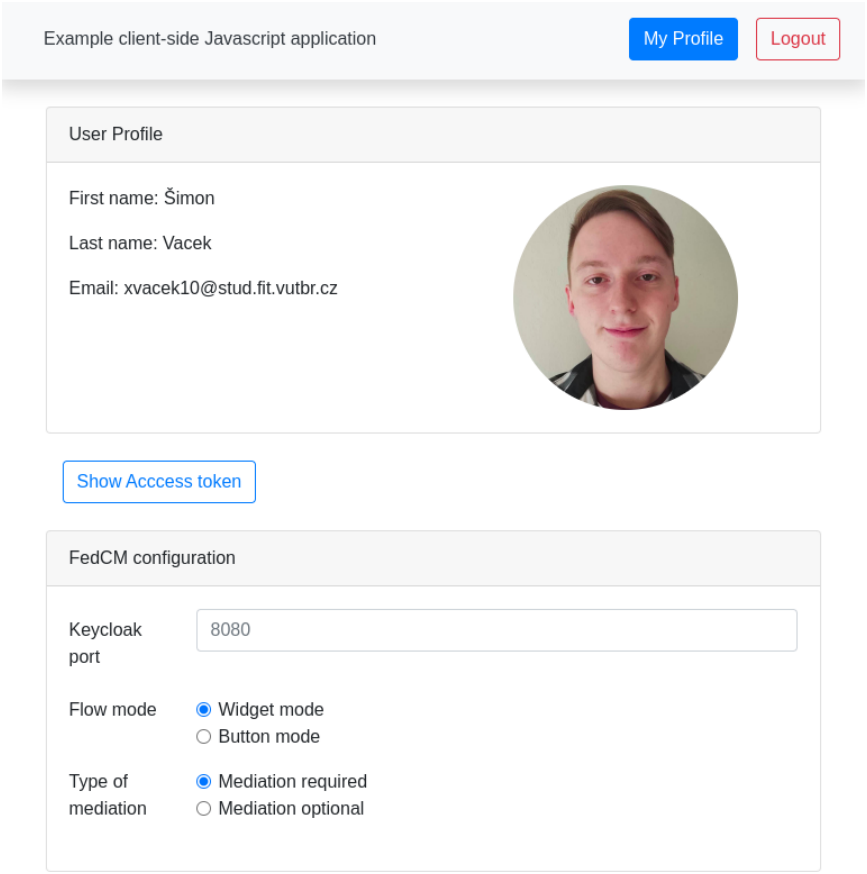


Figure 7.3: The client application used for testing displaying user information after a successful federated log-in.

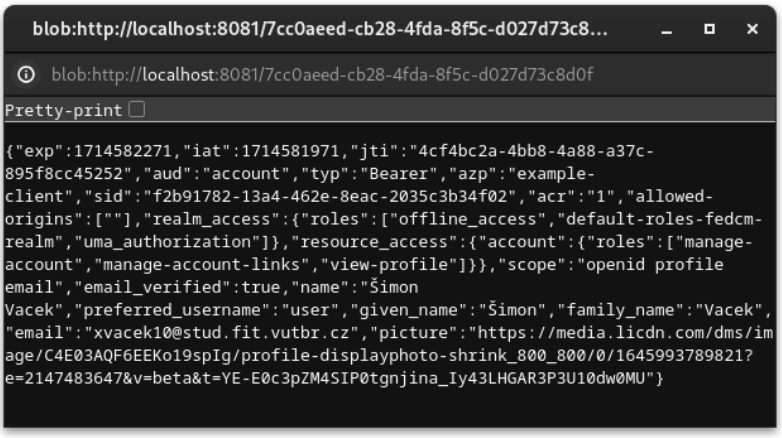


Figure 7.4: The access token obtained by the client application.

Chapter 8

Conclusion

The goal of this thesis was to study, implement, and integrate the Federated Credential Management API into Keycloak. This developing non-standardized specification, part of the Privacy Sandbox initiative, brings a solution to the third-party cookie phase-out in the second half of 2024. The same cookies being abused to track users are also used and relied on by identity providers to identify users and manage active sessions.

Before implementation, the current authentication and authorization protocols, with a focus on OpenId Connect and OAuth 2.0, had to be studied. The individual flows of the OIDC protocol, tokens enabling access to user's information, and resources protected by Keycloak in client applications provided the know-how of how Keycloak works and how the FedCM can be integrated along these protocols. The current solution in the Keycloak Javascript adapter mitigating the issues with blocking third-party cookies was explored. Two main features were identified to be affected the most – session management and silent authentication. Finally, to provide the full context, the proposed solution of FedCM API was explored in detail, along with other experimental features currently prototyped in the Google Chrome browser.

A solution was designed leveraging the Keycloak service provider interfaces, which allow easy extendability and the creation of custom REST endpoints. The final implementation satisfies the main requirements set by the specification and the extra Error API, which sends a message to a user when the login fails. The FedCM authorization extension is identified as a great addition to the specification and one of the first steps to be taken in future implementation in Keycloak.

Through the research of FedCM, it is clear that the specification tries to be independent of any other protocols to allow more freedom in implementation. However, this could lead to future revisions, adding more security layers incompatible with existing solutions in other protocols. This would result in re-inventing the wheel and ending up doing the opposite of what it tries to do: to reuse parts of what already exists as much as possible.

Future iterations of the Keycloak FedCM extension will explore the use of SAML clients with FedCM and integration with other standards built on top of OIDC and OAuth, such as the Proof Key for Code Exchange (PKCE) used in public clients and Demonstrating Proof-of-Possession (DPoP). Future experimentation and cooperation on the specification will tell whether FedCM can replace the Keycloak Javascript adapter or at least help mitigate the issues with the third-party cookie phaseout.

Bibliography

[1] *Federated Credential Management (FedCM)* [online]. 2024 [cit. 2024-04-03]. Available at: https://github.com/fedidcg/FedCM/blob/main/explainer.md.

[2] Google. *FedCM updates: IdP Sign-In Status API, Login Hint, and more* [online]. 2023 [cit. 2024-04-03]. Available at: https://developers.google.com/privacy-sandbox/blog/fedcm-chrome-116-updates.

[3] Google. *FedCM updates: Login Status API, Error API, and Auto-selected Flag API* [online]. 2023 [cit. 2024-04-03]. Available at: https://developers.google.com/privacy-sandbox/blog/fedcm-chrome-120-updates.

[4] Google. *Support auto-reauthentication in FedCM* [online]. 2023 [cit. 2024-04-03]. Available at: https://developers.google.com/privacy-sandbox/blog/fedcm-auto-reauthn.

[5] Google. *FedCM updates: Disconnect API and two updates* [online]. 2024 [cit. 2024-04-19]. Available at: https://developers.google.com/privacy-sandbox/blog/fedcm-chrome-122-updates.

[6] *Authorizing non-profile oauth scopes* [online]. 2024 [cit. 2024-04-20]. GitHub issue. Available at: https://github.com/fedidcg/FedCM/issues/477.

[7] Goto, S. and Moreno, N. P., ed. *Federated Credential Management API* [online]. Federated Identity Community Group and Google, 2024 [cit. 2024-03-22]. Available at: https://fedidcg.github.io/FedCM/.

[8] Hardt, D. *The OAuth 2.0 Authorization Framework* [online]. October 2012 [cit. 2024-03-22]. Available at: https://www.rfc-editor.org/rfc/rfc6749.

[9] Jones, M. and Hardt, D. *The OAuth 2.0 Authorization Framework: Bearer Token Usage* [online]. Microsoft and Independent, october 2012 [cit. 2024-03-22]. Available at: https://www.rfc-editor.org/rfc/rfc6750.

[10] Keycloak. *Keycloak Logos, Diagrams and more* [online]. 2024 [cit. 2024-04-19]. GitHub repository. Available at: https://github.com/keycloak/keycloak-misc.

[11] Keycloak. *Securing Applications and Services Guide* [online]. User manual, 24.0.0. March 2024. Available at: https://www.keycloak.org/docs/24.0.0/securing_apps.

[12] Keycloak. *Server Administration Guide* [online]. User manual, 24.0.0. March 2024. Available at: https://www.keycloak.org/docs/24.0.0/server_admin.

[13] Keycloak. *Server Developer Guide* [online]. User manual, 24.0.0. March 2024. Available at: https://www.keycloak.org/docs/24.0.0/server_development.

[14] Lodderstedt, T., McGloin, M. and Hunt, P. *OAuth 2.0 Threat Model and Security Considerations* [online]. Deutsche Telekom AG and IBM and Oracle, january 2013 [cit. 2024-04-20]. Available at: https://www.rfc-editor.org/rfc/rfc6819.

[15] Medeiros, B. de, Jones, M. B., Bradley, J., Sakimura, N. and Agarwal, N. *OpenID Connect Session Management 1.0* [online]. Google and Microsoft and NAT.Consulting and Yubico, september 2022 [cit. 2024-04-21]. Available at: https://openid.net/specs/openid-connect-session-1_0.html.

[16] Medeiros, B. de, Scurtescu, M., Tarjan, P. and Jones, M. B. *OAuth 2.0 Multiple Response Type Encoding Practices* [online]. Google and Facebook and Microsoft, 2014 [cit. 2024-03-22]. Available at: https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html.

[17] Muzikář, V. *Approvals System for Keycloak server*. 2018. [cit. 2024-04-22]. Master's thesis. Masaryk University, Faculty of Informatics, Brno. Supervisor Bayer, J. Written in Czech. Available at: https://is.muni.cz/th/y1txu/?lang=en.

[18] Sakimuraand, N., Bradley, J., Jones, M. B., Medeiros, B. de and Mortimore, C. *OpenID Connect Core 1.0* [online]. NAT.Consulting and Yubico and Self-Issued Consulting and Google and Disney, 2023 [cit. 2024-03-22]. Available at: https://openid.net/specs/openid-connect-core-1_0.html.

[19] Satragno, N., Hodges, J. and West, M., ed. *Credential Management Level 1* [online]. Web Application Security Working Group and Google, 2024 [cit. 2024-04-07]. Section 2.2. Available at: https://w3c.github.io/webappsec-credential-management.

[20] Thorgersen, S. and Silva, P. I. *Keycloak - Identity and Access Management for Modern Applications*. 2nd ed. Packt Publishing, 2023. ISBN 978-1-80461-644-4.

Appendix A

Federated Credential Management API Keycloak extension

This chapter describes the contents of the memory media submitted, the process of building, configuring, generating API documentation, and running both the Keycloak with the FedCM extension and the demo client application in two ways. The first is the preferred way to use the provided container images at the root of the directory. The second is using Maven and `http-server` node package, which could be unreliable and leave data after.

The submitted directory contains the source files, including the code and scripts necessary for the project and all the required project configurations. Additionally, it contains the Docker images for executing the project.

The implementation leverages the Keycloak service provider interfaces, which can be compiled and delivered separately. It, however, needs a Keycloak distribution to run. The Keycloak project is huge, takes a long time to compile, and could cause trouble with the packaging. Because minimal modifications were made to the base Keycloak codebase, Keycloak was compiled, packaged, and included in the `keycloak-dist` directory. The changes made in Keycloak are produced by `git diff commit2 commit1` and included in the file `keycloak.diff`. It can also be used to apply the patch on the Keycloak source code with `patch -p1 < path/to/keycloak.diff`.

The `data` directory contains the configuration for Keycloak, ready for a demo testing of the functionality. In `docs/javadoc/`, generated API documentation will be placed once generated, and `docs/report` holds the technical report, the thesis text.

The `docker` directory contains files for generating the Docker images for execution and manual testing. The `keycloak.tar` and `client.tar` archives are exported Docker images. They were generated by their Dockerfiles.

As mentioned, the Keycloak distribution is in `keycloak-dist`. This is used in case the project is executed with Maven. If it is, the built fedcm extension is placed inside it during the server's start.

The `mvnw` script was generated from the `pom.xml` file. The Maven wrapper script is the preferred method of building.

The `src` directory has the source code for both the main Keycloak FedCM extension, the object of this thesis, and a Javascript application for testing the FedCM functionality.

The diagram A.1 shows and describes the contents of the submitted media.

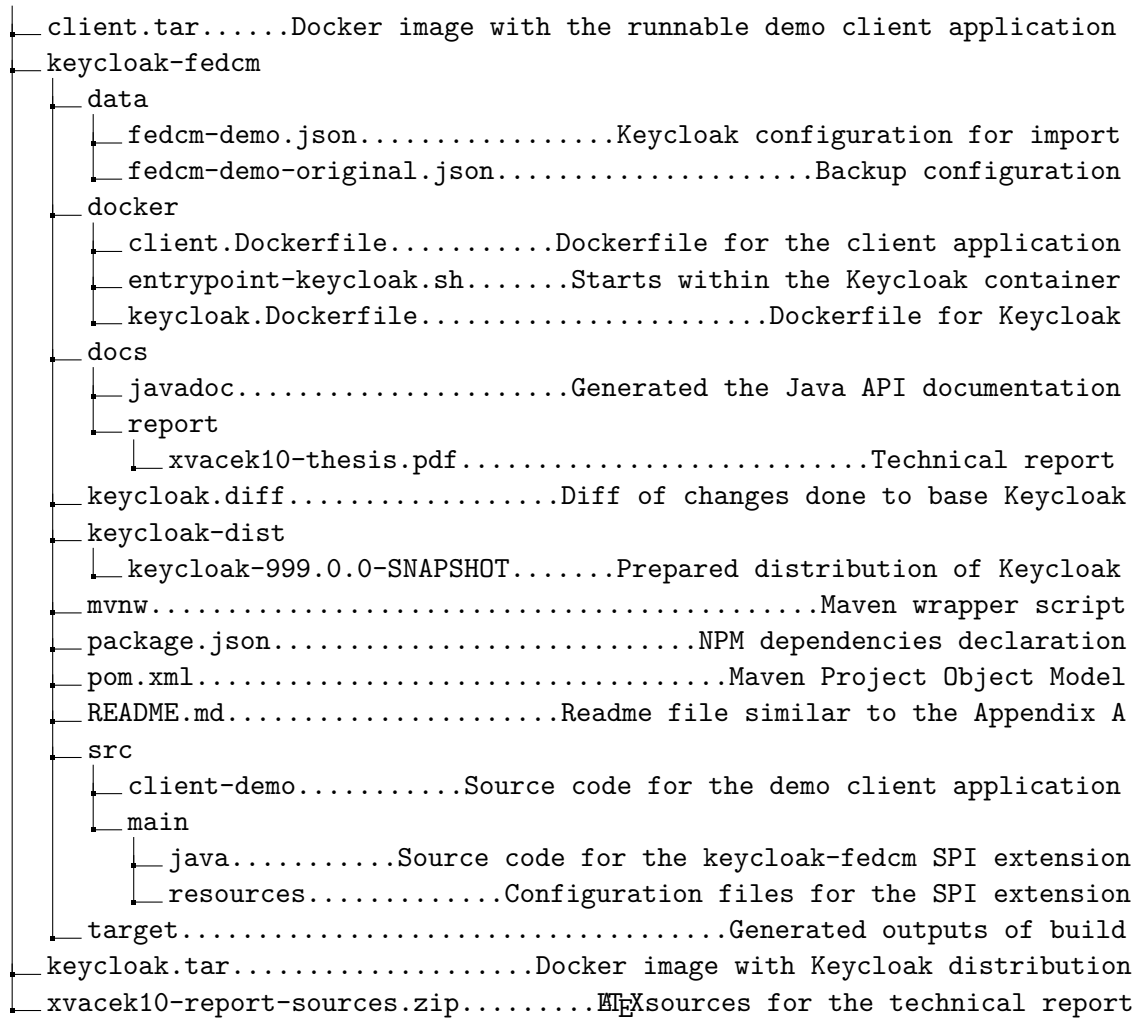


Figure A.1: Contents of the submitted directory. The `keycloak-fedcm` directory is decompressed `keycloak-fedcm` ZIP file on the media.

A.1 Prerequisites

This project MUST be built and run on Linux distributions as it relies on bash and other utilities.
OpenJDK 17 is recommended for building.
Maven wrapper requires the `JAVA_HOME` environment variable to be set.

A.2 Building the project

The FedCM extension to Keycloak is a Java Maven project. It uses a Maven wrapper for building and execution but is less preferred to Docker images.

A.2.1 Compile

To compile the extension and package it run:
`./mvnw exec:exec@compile`

A.2.2 Generate Javadocs

To generate the API documentation, run:
`./mvnw javadoc:javadoc`

A.3 Running the project

The project is run and tested locally. Both Keycloak and the client application must be hosted on **localhost**. The default port for Keycloak is `8180` and for the client application `8080`. After successful execution, open the client application on `localhost:8080` and follow the instructions there.

A.3.1 Running with Docker

This guide can be followed exactly step by step. **It is advised not to run the containers in detached mode**, especially for Keycloak, which runs a script depending on user input.

The provided images do not rely on Docker. A daemonless container tool, Podman, could be used as an alternative to Docker—if Podman is preferable, substitute `docker` for `podman` in these commands.

Running the client application

1. Load the Docker image:
   `docker load -i client.tar`

2. Set environment variable for the client application port. The application will be hosted on this port:
   `CLIENT_PORT=8080`

3. Run the container:
   `docker run -it --name client-app -p \$CLIENT_PORT:\$CLIENT_PORT -e CLIENT_PORT=\$CLIENT_PORT fedcm-demo-app`

If it is discovered the port for the client was wrong, the container needs to be removed and started again:

1. Stop the container:
   Press `Control+C` or `docker stop client-app`
2. Remove the container:
   `docker rm client-app`
3. Set environment variable for a different client application port:
   `CLIENT_PORT=8080`
4. Run the container:
   `docker run -it --name client-app -p $CLIENT_PORT:$CLIENT_PORT -e CLIENT_PORT=$CLIENT_PORT fedcm-demo-app`

The container can be stopped and run again:

1. Stop the container
   Press `Control+C` or `docker stop client-app`
2. Start the stopped container again
   `docker start -ia client-app`

Once the testing is done, remove the container and the image:

1. Remove the container
   `docker rm client-app`
2. Remove the image
   `docker image rm fedcm-demo-app`

Running Keycloak

Because the chosen default ports may not be available, the container starts an interactive script that lets the user configure the ports. Three actions are defined:

- `start` – This option starts Keycloak
- `reconfigure` – This allows changing the Keycloak configuration for the client application port. It presents the user with another choice:
  - *`port`* port of the client application
  - `default` restores the original configuration for Keycloak set in the Docker image.
  - *These options only change the configuration file. This configuration must be imported to work. They also modify only the original configuration file, meaning all other data stored in Keycloak is lost if it was not in the configuration already*
- `import` – This option imports the configuration file.

1. Load the Docker image:
   `docker load -i keycloak.tar`

2. Set environment variable for the port of Keycloak:
   `KEYCLOAK_PORT=8180`

3. Run the container:
   `docker run -it --name keycloak -p $KEYCLOAK_PORT:$KEYCLOAK_PORT -e KEYCLOAK_PORT=$KEYCLOAK_PORT keycloak-fedcm`

If it is discovered the port for Keycloak was wrong, the container needs to be removed and started again:

1. Stop the container:
   `Control+C` or `docker stop keycloak`

2. Remove the container:
   `docker rm keycloak`

3. Set environment variable for a different Keycloak port:
   `KEYCLOAK_PORT=8180`

4. Run the container:
   `docker run -it --name keycloak -p $KEYCLOAK_PORT:$KEYCLOAK_PORT -e KEYCLOAK_PORT=$KEYCLOAK_PORT keycloak-fedcm`

The container can be stopped and run again:

1. Stop the container:
   `Control+C` or `docker stop keycloak`

2. Start the stopped container again
   `docker start -ia keycloak`

Once testing is done, remove the container and the image:

1. Remove the container:
   `docker rm keycloak`

2. Remove the image:
   `docker image rm keycloak-fedcm`

A.3.2 Running with Maven

Running the client application

The client application does not use Maven. It is a plain HTML and Javascript application with some Bootstrap CSS. It needs to be served on localhost, and the default port for it is `8080`. The `http-server` package is used to serve the client. It can be downloaded from the dependencies.

1. Download the `http-server` package:
   `npm install`

2. Create an alias for the http-server executable:
   `alias http-server=node_modules/http-server/bin/http-server`

3. Run the server. Other ports than `8080` can be used, but it requires reconfiguring Keycloak, as described below.
   `http-server ./src/client-demo/ -p 8080`

Running Keycloak

The project needs to be built before executing the Keycloak extension. The Keycloak server requires importing a configuration file with realms, users, and clients for testing. Then, it can be run.

- To compile the project run:
  `./mvnw exec:exec@compile`
- The configuration file can be modified if the client application is running on a different port than `8080`:
  `./mvnw -Dclient.port=8080 exec:exec@reconfigure`
- If a mistake was made in the configuration, it can be restored:
  `./mvnw exec:exec@config-default`
- To import the configuration, run:
  `./mvnw exec:exec@import`
- This command then runs the Keycloak server. The variable for a port can be omitted for a default value or changed.
  `./mvnw exec:exec@start -Dkeycloak.port=8180`
- **Typically, the whole execution would look like this**:
  `./mvnw -Dclient.port=8080 -Dkeycloak.port=8180 exec:exec@compile exec:exec@reconfigure exec:exec@import exec:exec@start`

Appendix B

Testing manual

This manual gives instructions on how to verify the functionality of the FedCM extension via the client application. This manual is available on the client application at the root.[1]

B.1 Prerequisites

- Make sure to use Google **Chrome version 124**. Your current version can be found by entering „chrome://version“ in the search bar
- Enter „chrome://flags/“ in Chrome search bar and **set the following flags:**
  - #test-third-party-cookie-phaseout : Enabled
  - #fedcm-without-well-known-enforcement : Disabled
  - #fedcm-skip-well-known-for-same-site : Disabled
  - #fedcm-idp-signin-status-api : Enabled
  - #fedcm-error : Enabled
  - #fedcm-disconnect : Enabled
  - #fedcm-button-mode : Enabled
- **Enable Third-party sign-in** in settings.[2]
  Sometimes Chrome automatically blocks it for a specific address during testing. Mostly when a user cancels the sign-in dialog. If that happens, the address needs to be manually allowed to show third-party sign-in prompts.
- **Run Keycloak on localhost.** The port can be modified, but the default and preferred is **8180**.
- Make sure **this application is run on localhost on port 8080**. If this port can not be used, this client needs to be reconfigured in the Keycloak admin console.

[1] http://localhost:8080
[2] chrome://settings/content/federatedIdentityApi

B.2 Keycloak credentials

Keycloak is already pre-configured with a realm and two users. These are the credentials used for signing in to Keycloak:

- Keycloak account console[3]
  - Username: user
  - Password: password
- Keycloak admin console[4]
  - Username: admin
  - Password: admin

B.3 Instructions for testing FedCM functionality

Before proceeding, it is recommended to open a new browser tab so the instructions can be read through while following them. Open also the Keycloak account console and the Keycloak admin console. Ensure the admin console opens in the fedcm-realm and not the master realm.

It is also good to have the settings page open in case the third-party sign-in is disabled. The settings page is on chrome://settings/content/federatedIdentityApi.

B.3.1 First sign-up

1. Sign in to the Keycloak account console.
2. Navigate to the profile page on the client application by clicking the „My Profile“ button at the top of the page
3. *(Optional) Set the port for the Keycloak server.*
4. Do not set any other values in the form and click the „Sign in“ button.
5. In the opened widget, confirm two links for the privacy policy and terms of service are present.
6. Click Continue
7. If the sign-in is successful, the Sign in button changes to Sign out, and the user information is displayed.
8. Click on the „Sign out“ button.
9. The access token is now deleted, and the User Profile section is empty.

[3] http://localhost:8180/realms/fedcm-realm/account
[4] http://localhost:8180/admin/master/console/#/fedcm-realm

B.3.2 Following sign-ins

1. Sign in to the Keycloak account console.
2. Navigate to the profile page on the client application by clicking the „My Profile“ button at the top of the page
3. *(Optional) Set the port for the Keycloak server.*
4. In the configuration form, click the „Mediation optional“ option. Leave the rest of the options as they are.
5. Click on the „Sign in“ button.
6. No prompt for the user is shown, the user is authenticated with Keycloak, and the User Profile is displayed.
7. Verify there is a client session for „example-client“ by navigating to the Keycloak account console and clicking the „Applications“ item in the menu.
8. Navigate back to the client application and the profile page.
9. Click on the „Show Access token“ to display the JSON token in a new window.
10. The information in the User Profile section is parsed from the access token. To confirm the access token can be used for requests to Keycloak, click the „Send request to Keycloak“. This sends a request with the token to the OIDC userinfo endpoint and displays the JSON response.
11. Press the „Sign out“ button and sign in again. This time, the automatic re-authentication is disabled, and the widget prompts again for consent.
12. Press the „Sign out“ button.

B.3.3 Login status API and Button mode

1. If signed in the Keycloak account console, sign out.
2. Navigate to the profile page on the client application by clicking the „My Profile“ button at the top of the page.
3. *(Optional) Set the port for the Keycloak server.*
4. Make sure the Widget mode is chosen.
5. Click on the „Sign in“ button.
6. After a while, an error is returned. That is because Keycloak notified the browser that there is no active session in this realm.
7. Sign in to the account console.
8. From the flow modes, choose „Button mode“ and sign in.
9. In the modal window, pick the account that should have the same information shown in the account console. Sign out.

B.3.4 Dynamic sign in-flow

Before proceeding, it should be known that the current implementation does not inform the browser about a sign-in status if the session is terminated, only when sign-out is performed.

1. Sign in to the Keycloak account console.
2. Navigate to the admin console. Ensure you are in the „fedcm-realm“.
   (a) Click on „Sessions“ in the left menu.
   (b) On the first item, press the „account-console“ link in the column „Clients“
   (c) On the first item, press the three vertical dots and press „Sign out“.
   (d) You are now signed out of the account console, but Chrome is not notified.
3. Navigate to the profile page on the client application by clicking the „My Profile“ button at the top of the page.
4. *(Optional) Set the port for the Keycloak server.*
5. In the configuration form, choose „Widget mode“
6. Press „Sign in“
7. You are presented with a prompt to sign in to the identity provider because the browser expected a signed-in user. Press continue.
8. Sign in as a user in the account console in the pop-up window
9. Open the browser console of this pop-up window (by default it is by pressing F12 and clicking „Console“)
10. Type in *IdentityProvider.close()* and hit enter.
11. Continue in the widget flow.