Department of Computer Science
Faculty of Science
Palacký University Olomouc

MASTER THESIS

The implementation of a Single Sign-On

2022

Bc. Dominika Gajdová

Supervisor:
RNDr. Martin Trnečka, Ph.D.

Study program: Applied Computer Science, Specialization: Software Development

Bibliografické údaje

Autor:	Bc. Dominika Gajdová
Název práce:	Implementace služby jednotného přihlášení
Typ práce:	diplomová práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2022
Studijní program:	Aplikovaná informatika, Specializace: Vývoj software
Vedoucí práce:	RNDr. Martin Trnečka, Ph.D.
Počet stran:	39
Přílohy:	1 CD/DVD
Jazyk práce:	anglický

Bibliographic info

Author:	Bc. Dominika Gajdová
Title:	The implementation of a Single Sign-On
Thesis type:	master thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2022
Study program:	Applied Computer Science, Specialization: Software Development
Supervisor:	RNDr. Martin Trnečka, Ph.D.
Page count:	39
Supplements:	1 CD/DVD
Thesis language:	English

Anotace

Tato práce se zabývá implementací systému jednotého přihlášení. Jejím obsahem je především analýza a rozbor existujících protokolů, řešení, a dále návrh řešení vlastního. Součástí vlastního řešení je server spravující identity uživatelů, webový portál umožňující správu uživatelů a služeb a mobilní aplikace určená k verifikaci uživatele přes QR kód.

Synopsis

This thesis focuses on the implementation of a single sign-on system. Its contents include an analysis of existing protocols and solutions, along with a proposal of a custom solution. The custom solution contains a user identity management server, a web portal allowing for user and service management, and a mobile application that enables user verification via a QR code.

Klíčová slova: systém jednotého přihlášení; identita, autentizace, autorizace, OpenID Connect, SAML

Keywords: single sign-on, identity, authentication, authorization, OpenID Connect, SAML

I would like to thank my supervisor RNDr. Martin Trnečka, Ph.D for providing me with the opportunity to study an interesting topic and guiding me in the process.

I hereby declare that I have completed this thesis including its appendices on my own and used solely the sources cited in the text and included in the bibliography list.

date of thesis submission author's signature

Contents

List of Figures

1 Introduction

There has never been a time in history in which identifying people was more prominent than today. People in the middle ages did not live in million person cities. They did not communicate with people across the world very often, if ever. It was sufficient enough to know the people who are your neighbours or live in the same town. If someone attempted to pretend to be somebody else, they would have likely been quickly detected.

The same cannot be said about nowadays — the era of the Internet. Billions of people are intertwined by a large system of connected computer networks with unthinkable amounts of data transferring every second. It is important that there exists a way to prove you are who you claim to be and it is important that it is executed well. At least well enough — nothing is ever impenetrable and every solution has its strong and weak parts.

There are stories about stolen or easily guessed passwords that noticeably raised today's authentication standards. A need arises to memorize tens of different passwords for different services or, ideally, have another program memorize them for us. The practice of reusing the same password on every website with the hope that none of these websites ever become vulnerable is not optimal.

Single sign-on systems (SSO) are a great improvement in user experience and convenience. The user only needs to log in once and does not need to enter any more credentials when visiting another application in the same SSO system, noticeably saving the user's time. Having a set of services reuse the same user credentials without each one of them having to maintain their own authentication system is very helpful for the developers as well.

This thesis is divided into two thematically separated parts:

1. A theoretical introduction into the topic, its terminology and technologies. In this part, the terms identity, authentication, authorization will be introduced and existing solutions and protocols such as OAuth, OIDC and SAML will be explained and compared.

2. A practical implementation of a SSO. In this part, a custom solution will be described — the architecture, individual flows, challenges and suggested solutions.

1.1 Identity

What exactly is it that uniquely identifies a person? In the context of a citizenship, it might be their personal identification card number or their driver's license. In other contexts it might be entirely something else but there is a common pattern — identifiers that uniquely identify a person. If we take one or more unique identifiers and optionally some more information about a person, such as their name, e-mail or cell number (i.e. attributes), then we have just created an identity. *Identity* is a set of attributes that uniquely describe a person in a given context. These attributes can also be referred to as claims. From now on the thesis will be considering the context of the online world.

A person can have multiple identities. It entirely depends on the context and identifiers used. For example, user can have a work identity where they are uniquely identified by an employee number within the company they are working for, but then also have private personal identity that serves a different purpose. In the world of online services we often hear the word account and understand it loosely as something that is created after registration and holds all our information. An account is an application specific construct that holds or refers to an identity. [1]

The same identity can be used with multiple providers, as is the case with some of the popular sign-in options - Google, Facebook and Apple. If a user signs in to an application with one of the providers, (considering already having been previously registered) and then tries to sign-in with a different one, they will be notified that they have previously signed in with the same identity but a different provider.

1.2 Loginless

Some experimental approaches to identifying users without any personal identifiers have been tried. An online schedule making tool called Coursicle has implemeneted what they call a 'Loginless'. The idea behind it is using the method of browser fingerprinting:

> "Browser fingerprinting is a powerful method that websites use to collect information about your browser type and version, as well as your operating system, active plugins, time zone, language, screen resolution and various other active settings." [2]

Coursicle generates a Universal Unique Identifier (UUID) for a person. It is used as their UserID which then associates them with a particular browser based on a taken browser fingerprint. Therefore, a user visiting using their application will have an account created automatically while visiting and all their created schedules will be associated with that account without the user having to sign up. Naturally, mechanisms for syncability and restorability had to be introduced since many users own at least two devices. [3]

This kind of identity detection is not suited for most applications and there is a major future problem — browser fingerprinting is not 100% accurate and modern browsers are trying to essentially eliminate it. After all, it is a user tracking tool and an invasion of privacy.

1.3 Authentication

The process of proving one's identity is called *authentication*. Most commonly it is performed by entering the right combination of username and password which has been previously registered. The password in this case is the proof. There are many different ways and approaches to authenticating and they can be divided into categories called factors. These factors are usually referred to as:

- Something you know — A password or a secret code.
- Something you have — A smartphone or physical security key.
- Something you are — Fingerprints or face recognition.

Using only one of the mentioned factors is called a *Single Factor Authentication*. That has been the common practice until the last few years in which security has been more discussed and companies have started to require users to use at least one additional factor, essentially setting *Two Factor Authentication* (2FA) as the new standard. Combining two or more factors is also known as *Multi Factor authentication* (MFA) where 2FA is the base case of MFA.

If knowledge factor is omitted completely then it is known as *passwordless authentication*. This method has been gaining a lot of popularity lately because of the convenience of not having to remember credentials for every application the user has an account in. There are other protocol standards being developed that battle the password problem, most notably SQRL and FIDO.

1.4 Authorization

The process of verifying whether a user has access (i.e. privilege) to a requested resource is called *authorization*. For example, role-based authorization can be used to control access to resources only available for a particular user role. Policy-based authorization will enforce a defined set of requirements, such as being older than 18 policy will only grant access to users satisfying this policy requirement. Claims-based authorization will allow access if and only if the user's identity claims contain the required claim.

1.5 Naming conventions

It is common that every protocol uses different terminology to describe the same concept. For example, authorization server (OAuth2), identity server (SAML) and an identity provider (OIDC) are all referring to a server performing the authentication. The same applies to the term for a client application: client, relying party, service provider, client application, etc. In this thesis, when describing a particular protocol, its designated terminology will be used. Else the terms identity server and client service will be used throughout the text.

1.6 SSO

Single Sign-On is a feature which enables users to authenticate only once and have immediate access to other services within the same SSO system. This ability eliminates the need to authenticate to every service separately and remember the password for each of them, making this feature incredibly convenient for users.

Technically speaking, the idea is to use an identity provider. By delegating authentication to the identity server, a single point of identity is established. The services delegating authentication do not need to manage their own database of users and do not need to implement their own authentication system. Instead of logging to the service, the user actually logs in to the identity provider where a single session for the entire SSO is created.

An abstract flow description:

1. User visits a service that uses an identity provider and initiates a sign-in.
2. Identity provider checks if an SSO session already exists. If it does, the session is updated accordingly and the user is returned back to the service authenticated. If no session exists yet, the authentication process is started.
3. User is redirected to identity provider's login page and authenticates with a method required by the service.
4. After successful login, an SSO session is created on the identity provider's side and the user is redirected back to the service.

In the following chapters, the two most used identity protocols, which define a way of passing authentication information between the client service and the identity server, will be introduced: OIDC and SAML. The OIDC protocol is built on top of another important protocol called OAuth 2.0 which will be introduced first.

2 OAuth 2.0

A client service may need to access some protected resource from a third party service on behalf of a user. Before OAuth, the client would need to share their credentials for the third party service with the client service and the client service would use them to authenticate and access those requested resources. That comes with a plethora of problems such as credentials needed to be stored in clear text on the client side or a nonexistent practice of restricting the access to only predefined resources. For example, a user could use an application that would go through their e-mail messages, scan through them and collect all relevant e-mails. The only way to perform this would be to give the credentials to the application, essentially giving them access and all privileges to the e-mail account and therefore introducing a great security risk. The OAuth protocol has been designed to solve this exact problem.

The OAuth protocol defines four roles:

- Resource Server - Server that hosts the protected requested resource.
- Resource Owner - User or entity that owns the requested resource on the resource server.
- Client - The application wanting to access the protected resources on user's behalf.
- Authorization server - Server performing the resource owner authentication, requesting consent from the user and issuing the authorization grant. [4]

The naming of the OAuth 2.0 authorization server can be misleading considering it also performs authentication. The authentication server and the authorization server can be two separate servers or a single server performing both operations.

Figure 1 visualizes the abstract protocol flow and its individual steps:

1. Client application requests authorization from the resource owner (the user). This is usually executed by presenting a screen of requested privileges to the user. After user consents, an intermediary authorization grant is passed back to the client application signalizing that it has been successfully authorized.
2. Client application exchanges the acquired authorization grant for an access token. This access token does not hold any user information. Its purpose is solely to be used for authorization and the token format can be any string as long as it is not easily guessed (UUID or randomly cryptographically generated string).
3. Client application uses the access token to access protected resources on the resource owner's behalf.

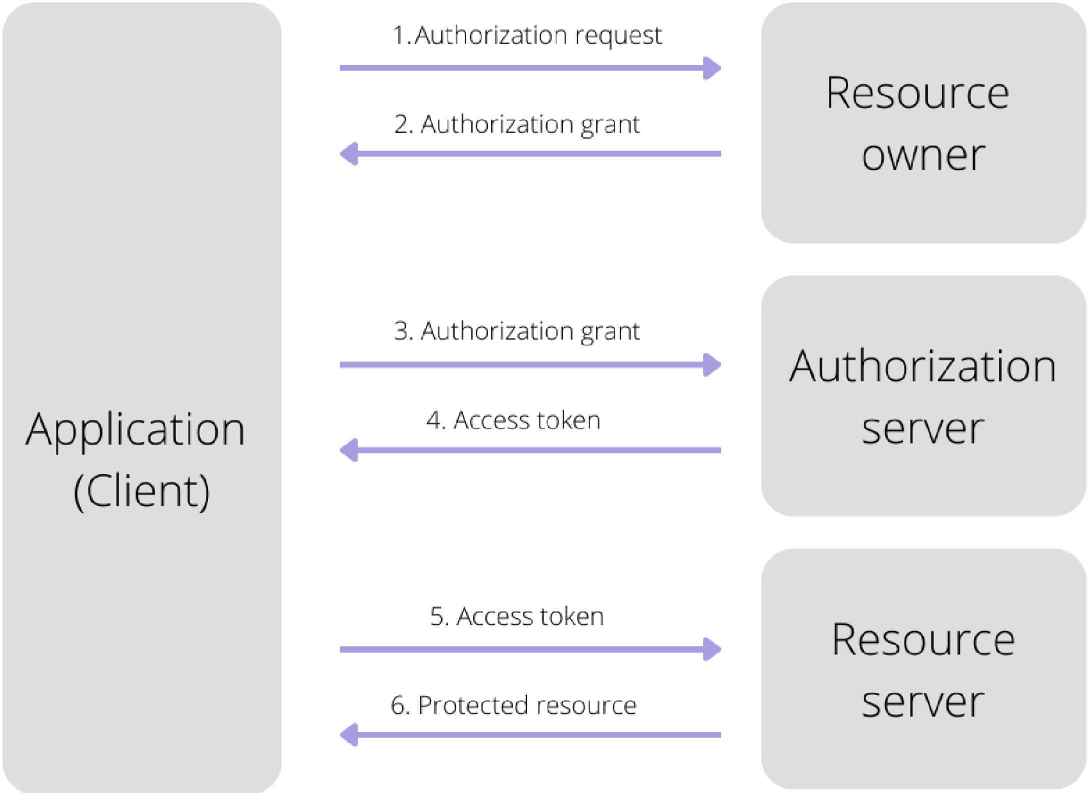


Figure 1: OAuth 2.0 abstract flow

Authorization endpoint

The are certain parameters that are being used with the OAuth 2.0 authorize endpoint: `GET /authorize`

- ClientId — Assigned unique identifier for the client application.
- Scope — Defines the scope of requested privileges, e.g. 'get:userInfo'.
- RedirectUri — Callback URL address that the authorization server sends the authorization grant or access token to.
- ResponseType — Grant type, usually either 'code' (authorization code) or 'token' (access token) or a combination of both.
- CodeChallenge — PKCE generated challenge.

Token endpoint

The token endpoint exchanges the authorization code for an access token: `POST /token`

- ClientId — Assigned unique identifier for the client application.

- RedirectUri — Callback URL address that the authorization server sends the authorization grant or access token to.
- Code — The authorization token received in the callback.
- CodeVerifier — PKCE generated verifier.

2.1 Authorization grant types

OAuth defines multiple grant types — different flows of obtaining an access token:

- Authorization code
- Implicit
- Client credentials
- Resource owner credentials

In the following subsections, the first two grant types will be explained as they are the most commonly used and relevant. The other two grant types can be studied further in the official RFC 6749 document. [4]

Implicit grant

The implicit grant flow was designed for use with front end clients such as single page applications or mobile applications. It was created at a time when CORS was not yet implemented by most browsers and public clients were not able to perform requests outside of their own domain, therefore making it impossible to exchange authorization code for an access token. [1]

1. Client requests authorization from the authorization server by redirecting to the authorize endpoint.
2. User is presented with a consent screen and is prompted to authenticate with the authorization server.
3. When the user successfully authenticates and consents, the authorization server redirects to the client's registered callback URL with the access token in a form of a URL hash fragment.
4. Client can now access protected resources on the user's behalf.

Receiving access code in URL is prone to interception attacks and therefore the implicit flow has important security issues to consider. Because CORS is now supported by almost all browsers, the implicit grant flow has been deprecated.

Authorization code grant

This flow directly implements the abstract authorization flow.

1. Client requests authorization from the authorization server by redirecting to the authorize endpoint.
2. User is presented with a consent screen and is prompted to authenticate with the authorization server.
3. When user successfully authenticates and consents, the authorization server redirects to the client's registered callback URL with the authorization code.
4. Client exchanges the authorization code for an access token and can now access protected resources on user's behalf.

The authorization code grant was designed to be used with a back end channel because the token exchange usually requires a client secret that needs to be stored securely. Front end clients cannot store secrets securely and therefore developers very often used an implicit flow.

Authorization code grant with PKCE

If the authorization grant flow is used in a public client context, it suffers from the same security problem as the implicit flow — the authorization code can be intercepted and exchanged for an access token by a malicious third party. A verification mechanism has been introduced to ensure that the application requesting the token exchange is the same as the application that originally requested authorization. This mechanism substitutes a public client secret and adds two parameters to the flow:

- CodeVerifier — Randomly cryptographically generated code. This parameter is passed to the token endpoint.
- CodeChallenge — T(CodeVerifier) where T is a hash function (usually sha256). This parameter is passed to the authorization endpoint.

The authorization server applies the same hash function to the CodeVerifier and compares the output with the saved CodeChallenge. This prevents an interceptor from being able to exchange the authorization code without also knowing the CodeVerifier (the proof). Both PKCE parameters can be generated on the front end side and are only temporarily held in memory, making this method sufficiently secure. [5]

2.2 Refresh token

Issuing an access token without an expiration date is a great security risk. In case it is compromised, the attacker gains unlimited access to a protected resource. Therefore, access tokens should be short lived. As a matter of fact, the expiration time is recommended to be only 15 minutes. It is not very convenient for the user though, as they would be asked to re-authenticate every 15 minutes. The refresh token mechanism solves this issue by introducing a possibility of obtaining a new access token without the user having to authorize again.

The typical Token endpoint response contains these four parameters:

- Access Token — The token used for authorization.
- Refresh Token
- Token Type — Type of issued token, e.g. Bearer.
- Expires In — Expiration of the access token in seconds.

Refresh token's purpose is to be exchanged for a new access token. It has longer expiration date. Depending on the use case and how often the user needs to authorize the client application, it could be anywhere between a day and a year. The shorter, the more secure, the less convenient for the user. Refresh token rotation ensures every refresh token is used only once and every additional attempt is flagged as a potentially compromised token.

Refresh tokens should be stored securely which can be quite the challenge with public clients. The universally most secure solution is to have the refresh token stored in an http only cookie. This cookie is set by the server and not accessible by client code, making it resistant to XSS attacks. This works well for services hosted on the same domain as the identity server but it is becoming very hard, if not impossible, for cross domain requests. Google has announced its intentions of blocking all third party cookies (i.e. cookies from other domains than the client's) by 2023 [6], joining Safari in the fight for users' enhanced privacy.

3 OpenID Connect

OAuth 2.0 was designed solely for authorization purposes. The OpenID Connect (OIDC) adds an identity layer on top of the OAuth 2.0 protocol which enables client applications to have users authenticated by the authorization server (or Identity Provider in OIDC terminology). Instead of a simple access token, a secure ID Token is issued to the client application, containing user information in the form of claims. The OIDC also introduces a UserInfo endpoint which provides another standardized way of obtaining user claims. The Identity Provider can also issue an access token or a refresh token. The access token can be used to obtain claims from the UserInfo endpoint. [7]

OIDC enables SSO functionality out of the box. Therefore, an implementation of an SSO solution can be reduced to an implementation of an OIDC Identity Provider.

The OIDC defines three roles:

- End User — The user being authenticated.
- Relying Party — An OAuth 2.0 client service that authenticates end user with the OpenID Provider and acquires claims about the authenticated user.
- OpenID Provider — The OAuth 2.0 authorization server which implements the OIDC protocol. [1]

3.1 Authorization Flows

Considering the OIDC is just a layer on top of OAuth 2.0, the implicit and authorization code flows share significant resemblance and only introduce a few key differences. The two most significant ones are:

1. The introduction of the ID token.
2. A login session being established by the OIDC Provider during the user authentication. This is the most important addition because this part directly enables an SSO.

Due to the infrequent use in practice, the hybrid flow will be introduced only briefly.

Implicit Flow

The implicit flow introduces a new response type — the ID Token. The flow remains the same as the implicit grant flow, only an ID Token is passed to the callback URL instead of the access token. The absence of passing an authorization grant in a URL hash fragment eliminates the access token interception problem, making this flow acceptable to use security-wise considering the ID Token does not contain any sensitive information.

Authorization Code Flow

The authorization code flow can have three possible response types - an ID Token, an access token and optionally a refresh token. PKCE should be used with public clients as well. Among others, the nonce attribute has been added to the authorization request as another security option. The nonce value should be random and is used to associate a client session with the ID Token and eliminate replay attacks.[1] Client service should generate it, store it and pass in to the authorize endpoint. OIDC Provider will add the nonce as a claim to the ID Token and the client service should verify a match after receiving it. In public clients, it is still preferred to use the PKCE mechanism or a combination of both.

Hybrid Flow

The hybrid flow combines elements from both the implicit flow as well as the authorization code flow. After client successfully authenticates, the browser redirects to the client callback URL with an ID Token and an authorization code. The client should validate the ID Token and call its own back end to exchange the authorization code to obtain additional tokens.

3.2 Single logout

In order to terminate an SSO session, the SSO cookie needs to be invalidated and all authorized client services need to be notified about the event. OpenID recognizes two different types of logout mechanisms: Back-Channel Logout and Front-Channel Logout.

Back-Channel Logout

The back-channel logout mechanism provides a secure channel between the relying party (back end of a service) and the OpenID provider to be used for communicating logout requests. It is more reliable compared to front-channel because in order for the front-channel to work, the web page needs to be opened in the browser to be notified about the logout. On the other hand, back-channel does not have access to browser storage and clearing the session is more demanding that simply clearing out session cookie or local storage. The cleaning process is left to the relying party to implement in its own way upon receiving a logout prompt from the OpenID provider. [8]

Front-Channel Logout

Front-Channel logout mechanism uses the browser's user agent for communication between the relying party and the OpenID provider. The relying party registers a logout URL that will be used for receiving a notification about the logout prompt. OpenID provider keeps track of all currently authenticated services in the session. During the logout process, the OpenID provider renders iframes with the session's authenticated services' logout URLs in a page. This

[1] Replay attack is kind of a man in the middle attack in which a request can be intercepted, delayed and resent again under the impression of a authentic message.

action notifies each relying party about the logout event and is requested to clear all state associated with the session. However, since most browsers have started to block 3rd party cookies, the logout URL might not be able to access the relying party's login state and therefore trigger the logout notification, because the iframes have a different origins than the OpenID provider. [9]

3.3 JWT

The introduction of the ID Token poses a question of what format should the ID Token use. This question led to a solution, developed by Microsoft, that has become widely used. ID Tokens are encoded into JWT format. JWT is a secure, compact, self-contained set of claims encoded into a JWS (JSON Web Signature) or JWE structure. These claims are expressed as a set of key-value pairs. JWT tokens are digitally signed either symmetrically using a secret (HMAC) or with a public/private key pair (RSA). Therefore, it is easy to verify the integrity of the claims contained in the token and it cannot be forged. JWT tokens can be encrypted (JWE), making them publicly unreadable. [10]

JWT is composed of three parts: header, payload and signature.

Header

Source code 1 shows the base JWT header example.

```
{
   "typ": "JWT", //defines type of object
   "alg": "HS256" //signature algorithm
}
```

Source code 1: JWT Header

Payload

Payload contains the claims. There are three types of claims: registered, public, and private. Registered claims are predefined and are recommended to be present, although not required by JWT standards — iss, sub, aud etc. Public claims are custom claims that can be used publicly and need to be registered in the IANA JSON Web Token Claims Registry. Private claims are custom claims that have specific meaning between a set of agreeing clients. An example payload can be seen in Source code 2.

```
{
   "iss": "https://identity-provider.com", //URL of the token issuer
   "sub": "123456789", //unique user identifier
   "aud": "86eadb58-1180-49d1-8e00-bd580a428b40", //Client ID
   "exp": "1656668336", //ID token expiration date in epoch
      timestamp format
   "iat": "1656668300", //date when ID Token was issued in epoch
      timestamp format
   "role": "admin" //private claim
}
```

Source code 2: JWT Payload

Signature

Signature is created by combining base64 encoded header, base64 encoded payload and secret/private key as demonstrated in code snippet 3.

```
HMACSHA256(
   base64UrlEncode(header) + "." +
   base64UrlEncode(payload),
   secret
)
```

Source code 3: JWT Signature

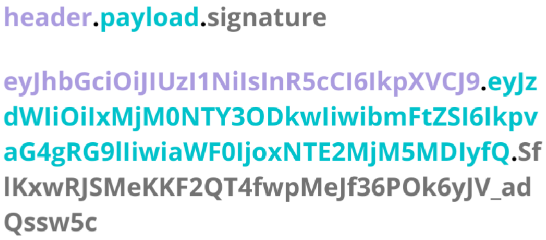
The final product

The final JWT token is composed of a base64 encoded header, base64 encoded payload and base64 encoded signature, all parts divided by a dot. The format, showcased in Figure 2, is visibly very compact (as opposed to SAML) and can easily be worked with HTTP/HTML environments.

JWT has become popular with APIs because it scales really well. There is no need to check database in order to verify user because the authentication state has been moved to the front end and only the signature needs to be checked.

4 SAML 2.0

The SAML 2.0 was introduced in 2005 as the first cross domain SSO solution. It is a protocol for authenticating web applications. Prior to that, SSO was only possible with websites hosted by the same domain, using a shared session cookie. SAML has been widely adopted, most notably in enterprise systems, because it provided an efficient way for companies to centralize identity management across



```
header.payload.signature

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJz
dWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpv
aG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.Sf
lKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_ad
Qssw5c
```

Figure 2: JWT format

multiple enterprise applications. [1]

Apart from cross-domain SSO, SAML's second feature is identity federation. Users can have different local identities for different services and the identity federation is an agreement between different parties on a particular identifier (e.g a user e-mail) that will be used to identify a user between all of them.

The SAML 2.0 identity protocol works in a very similar way to the OIDC. A user visits an applications, initiates a log in, SAML request is created and the user is redirected to the identity provider's login page. After authenticating, the identity provider generates a SAML Assertion and the user is redirected back to the original application, now authenticated. If the user was already authenticated, the login page is skipped.

SAML is XML based which makes it less attractive for developers and much more difficult to implement in contrast to REST API friendly OIDC. Nonetheless, SAML is still a valid protocol and is being used today even though OIDC has been the preferred solution for new systems.

SAML's defined main roles:

- Subject — The authenticated user whose identity information will be exchanged.
- SAML Assertion — A claims-based XML message containing user identity information passes via HTTP redirects.
- Identity Provider — The authentication server which issues SAML Assertions about an authenticated user.
- Service Provider — The client service requesting user authentication. [11]

4.1 Flow

Two SAML flows are defined depending on the service that originates the authentication process. If the process is started by a user visiting a service provider, it is called a Service Provider Initiated (SP-Initiated) SSO. If the user visits the identity provider's portal dashboard, for example, and accesses other service providers from there, it is called a Identity Provider Initiated (IdP-Initiated) SSO. [1]

SP-Initiated

The user starts at the service provider.

1. User visits a service provider — an application they want to access.
2. Service provider generates a SAML request and redirects to the identity provider with the request attached.
3. Identity provider checks if the user is already authenticated. If they are, further authentication is skipped. Otherwise, user is redirected to the identity provider's login page where they are provided a means of logging in.
4. After the user successfully authenticates, the identity provider generates a SAML response which contains a SAML Assertion — the user's identity information. The response is passed to an Assertion Consumer Service URL, which is a registered URL designed to accept SAML responses (SAML analogy to OAuth authorize callback URL).
5. Service provider validates and parses the SAML XML message and considers the user authenticated.

IdP-Initiated

The user starts at the identity provider. The identity provider initiated flow is typical for enterprise systems, where user logs into a portal which contains links to other service providers. Therefore, the service provider receives a SAML response without creating the SAML request in the first place.

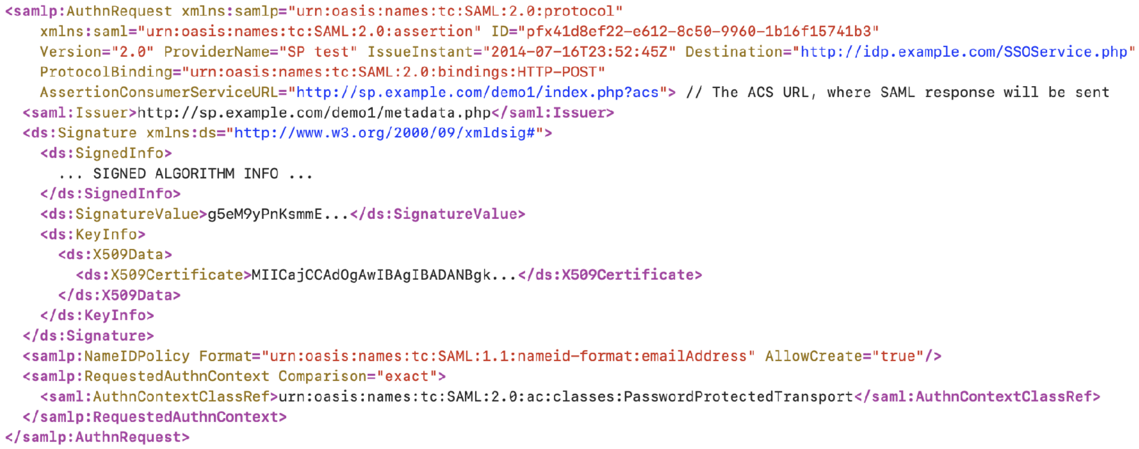
1. User visits an identity provider's login page or portal dashboard.
2. In case of a portal, it creates a SAML request and redirects the user to the identity provider with the request attached.
3. Identity provider checks if the user is already authenticated. If they are, further authentication is skipped. Otherwise, user is redirected to the identity provider's login page where they are provided a means of logging in.
4. Identity provider redirects the user back to the portal, passing in the SAML response. User is now authenticated in the portal and can access a list of available applications they can access.

5. User tries to access an application from the portal list. The browser redirects to the identity provider with a piece of information about the service trying to authenticate. The identity provider checks for a valid session.

6. Considering the session is valid, the identity provider redirects to the service provider's Assertion Consumer Service URL, passing in the SAML response. The user is now authenticated with the service provider.

4.2 Example

The request signature can either be passed separately via HTTP-Redirect binding or the signature can be embedded into the request as seen in Figure 3.



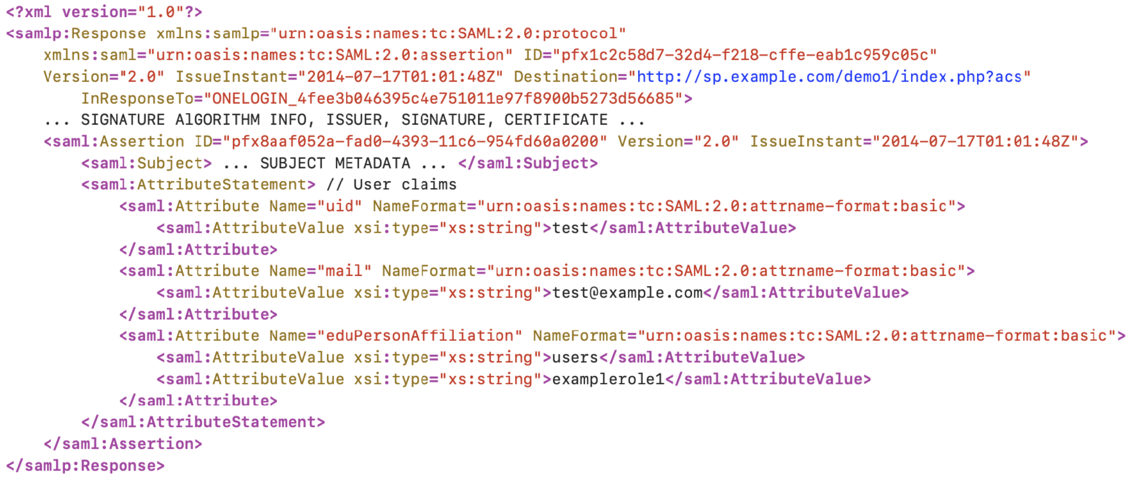
```
<samlp:AuthnRequest xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" ID="pfx41d8ef22-e612-8c50-9960-1b16f15741b3"
    Version="2.0" ProviderName="SP test" IssueInstant="2014-07-16T23:52:45Z" Destination="http://idp.example.com/SSOService.php"
    ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
    AssertionConsumerServiceURL="http://sp.example.com/demo1/index.php?acs"> // The ACS URL, where SAML response will be sent
  <saml:Issuer>http://sp.example.com/demo1/metadata.php</saml:Issuer>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
      ... SIGNED ALGORITHM INFO ...
    </ds:SignedInfo>
    <ds:SignatureValue>g5eM9yPnKsmmE...</ds:SignatureValue>
    <ds:KeyInfo>
      <ds:X509Data>
        <ds:X509Certificate>MIICajCCAdOgAwIBAgIBADANBgk...</ds:X509Certificate>
      </ds:X509Data>
    </ds:KeyInfo>
  </ds:Signature>
  <samlp:NameIDPolicy Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress" AllowCreate="true"/>
  <samlp:RequestedAuthnContext Comparison="exact">
    <saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport</saml:AuthnContextClassRef>
  </samlp:RequestedAuthnContext>
</samlp:AuthnRequest>
```

Figure 3: SAML Request Example from samltool.com

The SAML Response format is the same. In this case, the issuer is the identity provider and an assertion message is attached, containing the audience, user claims and session metadata. Figure 4 displays a simplified SAML response.

OIDC vs SAML 2.0

SAML and OIDC are two very similar protocols. The most noticeable differences are the format and terminology used. OIDC uses a compact JSON format, SAML is XML-based. OIDC has authorization built in whilst SAML only focuses on authentication and lets the system implement their own authorization process. As already mentioned, OIDC is much more popular nowadays because it is relatively easy to implement and its restful design is very developer friendly. Configuring SAML can be demanding and time-consuming but it does deliver results.



```xml
<?xml version="1.0"?>
<samlp:Response xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" ID="pfx1c2c58d7-32d4-f218-cffe-eab1c959c05c"
    Version="2.0" IssueInstant="2014-07-17T01:01:48Z" Destination="http://sp.example.com/demo1/index.php?acs"
        InResponseTo="ONELOGIN_4fee3b046395c4e751011e97f8900b5273d56685">
    ... SIGNATURE AlGORITHM INFO, ISSUER, SIGNATURE, CERTIFICATE ...
    <saml:Assertion ID="pfx8aaf052a-fad0-4393-11c6-954fd60a0200" Version="2.0" IssueInstant="2014-07-17T01:01:48Z">
        <saml:Subject> ... SUBJECT METADATA ... </saml:Subject>
        <saml:AttributeStatement> // User claims
            <saml:Attribute Name="uid" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                <saml:AttributeValue xsi:type="xs:string">test</saml:AttributeValue>
            </saml:Attribute>
            <saml:Attribute Name="mail" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                <saml:AttributeValue xsi:type="xs:string">test@example.com</saml:AttributeValue>
            </saml:Attribute>
            <saml:Attribute Name="eduPersonAffiliation" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                <saml:AttributeValue xsi:type="xs:string">users</saml:AttributeValue>
                <saml:AttributeValue xsi:type="xs:string">examplerole1</saml:AttributeValue>
            </saml:Attribute>
        </saml:AttributeStatement>
    </saml:Assertion>
</samlp:Response>
```

Figure 4: SAML Response Example from samltool.com

5 Existing identity solutions

Existing solutions make the task of implementing an identity provider easier for developers. If specific custom behavior is not needed and the budget allows it, they are the perfect option for businesses and individuals. Most of these solutions are targeting enterprises and offer a very wide variety of features. Often, no additional code is needed since all configuration can be set up in the solution portal and prepared libraries and SDKs are available. Three out of many popular identity providers will be described and compared: Okta, Auth0 and Keycloak.

5.1 Okta

Okta is a giant cloud-based identity solution and proudly proclaims to be the number one identity platform in the world. Okta offers over 7000 different integrations making it very flexible to integrate with almost any other service. It offers products ranging from SSO, MFA, authentication, authorization, user management to some more advanced features such as API access management or a passwordless login solution. It also supports 3rd party integrations making the possibilities practically endless.

Regarding SSO, Okta offers integrations with all standard protocols such as OAuth 2.0, SAML 2.0, WS-Fed and more. Okta implemented its own SSO solution called Secure Web Authentication (SWA) which is targeted at services that do not support previously mentioned federated protocols. Desktop SSO is also available with products called IWA and Agentless.[12]

Another important Okta product is called Okta Fastpass. It provides passwordless authentication across the system by registering user devices (laptops, mobile phones etc.) and creating a binding between the device and Okta. Multiple 2FA methods are available including e-mail magic links, WebAuthn or Okta's own Verify mobile application which enables users to choose their preferred 2FA option: push notification, a temporary code or biometrics. [12]

With many great features and practically endless possibilities of integrations, Okta is a great fit for enterprises and businesses on the market and it is reflected in the pricing plan as well.

5.2 Auth0

Auth0 is another cloud-based identity solution on the market. It focuses more on developers and thus it is not as modular and integration rich as Okta. All the core functionality is available including user management, authentication, authorization, MFA, social logins, passwordless and SSO. Auth0 offers a subset of features that Okta does.

Auth0 offers more advanced MFA tools such as adaptive MFA [2] or MFA that can be prompted when accessing a resource that requires stronger authorization.

It is a great and budget friendly product mainly targeted at development teams and smaller businesses, depending on the use case. It also offers a free version which is limited by the size of the user base which makes Auth0 a popular choice. Auth0 is not suitable for enterprise solutions considering the user base is not as scalable compared to Okta.

5.3 Keycloak

Keycloak is worth mentioning because it is a great open-source solution with no extra cost. Keycloak is an identity an access management solutions offering all the core features out of the box including SSO, social logins, OIDC, SAML 2.0 and OAuth 2.0 integrations and more. It is a great alternative if the use case allows the hosting of the Keycloak identity server instead of using a cloud-based solution.

5.4 Summary

All mentioned solutions offer core identity solution features and a user/admin panel UI which lets customers manage services and set up feature configurations. For development teams that are willing to host the identity server themselves, Keycloak is a great choice. For cloud-based solutions, Okta or Auth0 will work well, depending on the size of the user base.

[2] Adaptive MFA refers to MFA being triggered only when a certain risk factor arises. For example, user could be asked to use another login factor when logging in from another country.

6 The implementation

The implementation part of this thesis revolves around a custom OIDC-inspired SSO system called KMI SSO which was created for a smaller user base. It is focused mainly on authentication and does not provide flexible authorization options — it is left to the developers to create their own authorization system (similar to SAML). The SSO system supports both IdP-initiated and SP-initiated authentication flows as described in the SAML section. The SSO system is composed of three products:

1. Identity server that provides central user management and authentication.
2. A portal which provides user interface for the authentication process and user panel for managing user account, services and users.
3. A verification mobile application (KMI Auth) that enables users to authenticate using a QR code.

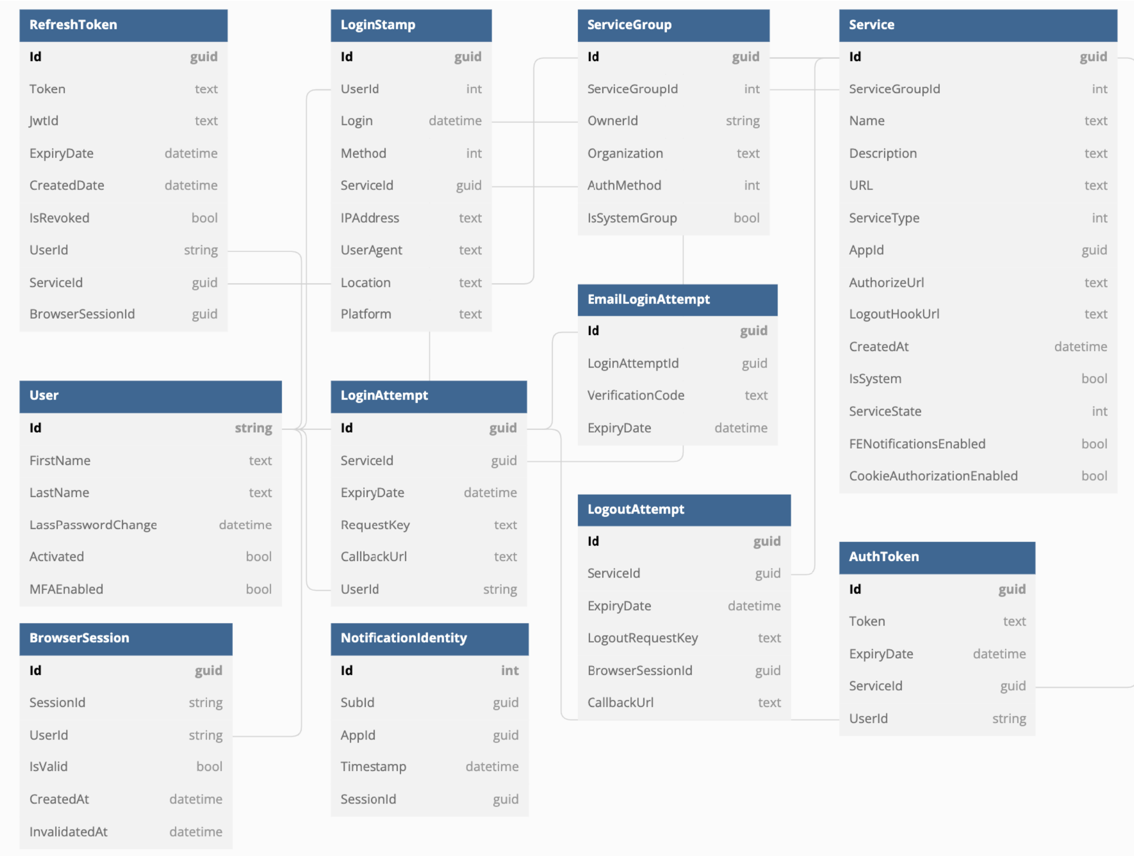
The identity server is created with the ASP.NET Core framework 6.0 with an open source object-relational database called PostgreSQL. The portal application is a single page application (SPA) created with NextJS framework. The mobile application is built with React Native — a multi-platform tool which enables the creation of native mobile applications (iOS, Android) using Javascript and React library. Additionally, a React Native development tool called Expo was used to simplify the process of development and testing. Expo provides its own mobile application which communicates with an Expo server hosted locally during development and enables hot reload functionality and platform independent testing without any complicated configuration.

6.1 Identity Server

Identity server establishes a single point of identity for registered users. It handles the authentication, user management, MFA and provides a public API for developers who wish to use the KMI SSO system.

Figure 5 models the core database relations used by the identity server. It is included for reference purposes considering the following identity server analysis contains references to these database entities.

Project-wise, the identity server uses Clean Architecture. Clean Architecture focuses on separation of concern principle and divides the application into separate layers that can be visualized by a series of nested circles as demonstrated in Figure 6. The figure also shows the inwards dependency flow — the inner most layer has no dependencies, the layer above is dependent on the layer below and so on.



RefreshToken	
Id	**guid**
Token	text
JwtId	text
ExpiryDate	datetime
CreatedDate	datetime
IsRevoked	bool
UserId	string
ServiceId	guid
BrowserSessionId	guid

LoginStamp	
Id	**guid**
UserId	int
Login	datetime
Method	int
ServiceId	guid
IPAddress	text
UserAgent	text
Location	text
Platform	text

ServiceGroup	
Id	**guid**
ServiceGroupId	int
OwnerId	string
Organization	text
AuthMethod	int
IsSystemGroup	bool

Service	
Id	**guid**
ServiceGroupId	int
Name	text
Description	text
URL	text
ServiceType	int
AppId	guid
AuthorizeUrl	text
LogoutHookUrl	text
CreatedAt	datetime
IsSystem	bool
ServiceState	int
FENotificationsEnabled	bool
CookieAuthorizationEnabled	bool

EmailLoginAttempt	
Id	**guid**
LoginAttemptId	guid
VerificationCode	text
ExpiryDate	datetime

User	
Id	**string**
FirstName	text
LastName	text
LassPasswordChange	datetime
Activated	bool
MFAEnabled	bool

LoginAttempt	
Id	**guid**
ServiceId	guid
ExpiryDate	datetime
RequestKey	text
CallbackUrl	text
UserId	string

LogoutAttempt	
Id	**guid**
ServiceId	guid
ExpiryDate	datetime
LogoutRequestKey	text
BrowserSessionId	guid
CallbackUrl	text

AuthToken	
Id	**guid**
Token	text
ExpiryDate	datetime
ServiceId	guid
UserId	string

BrowserSession	
Id	**guid**
SessionId	string
UserId	string
IsValid	bool
CreatedAt	datetime
InvalidatedAt	datetime

NotificationIdentity	
Id	**int**
SubId	guid
AppId	guid
Timestamp	datetime
SessionId	guid

Figure 5: Database schema (excluding 1:M, M:M relationship tables)

Domain layer contains the enterprise logic — entities, enums, interfaces and enterprise-specific logic — and has no dependencies. Application layer contains business logic and is only dependent on the Domain layer. The application layer defines interfaces that are implemented by services in the Infrastructure layer. The Infrastructure layer contains services based on interfaces from Application layer which provide access to external resources such as database or SMTP. Presentation layer contains the user interface of the application or a web API depending on the intended functionality.

Clean architecture is very flexible as it allows developers to postpone technology decisions for a later time. Because the domain and application layers are only dependent on interfaces, the interface implementations can be changed at any time and the enterprise and business logic will not be affected. The architecture is also independent of the presentation layer which can be replaced seamlessly.

The system primarily uses role-based authorization and defines three user roles:

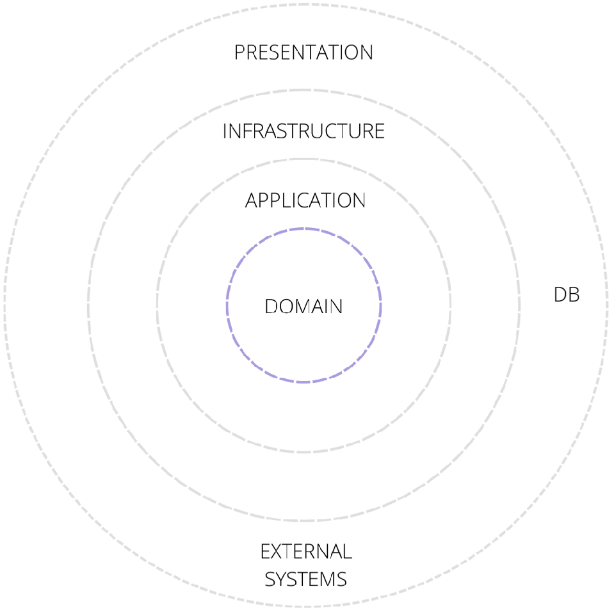


Figure 6: Clean Architecture layer structure

- User — A regular user that has access to services within service groups they are a part of. The user role is the most limited.
- Service Owner — User can manage service groups, services and their corresponding users.
- Admin — User can manage service owners and services (including system services, such as mobile app or front end client). Admin is responsible for approving or denying a service and service manager registration requests.

All roles have access to their account management, where they can change their password and profile information.

Additionally, a policy-based mechanism is used to determine whether a service has access to private API endpoints. The policy requirement checks whether the requesting service is truly a system service (an internal SSO system service).

Each service owner account comes with a space called *service group*. The service group contains all services and users between which an SSO is established. The initial trust between the SSO system and the client service is based upon an SSO admin approval. When a service manager registration is completed, the account is not yet active and has to be approved by an admin user first. The same applies for service registration. Registered services which are awaiting approval are referred to as pending services. Service managers will receive an informative e-mail regarding a rejection/approval of a pending service.

The SSO service recognizes three types of client services that a service manager can register:

- Website — A web service (either a back end server or a front end application).
- Mobile — A native mobile application.
- Desktop — A native desktop application. In the current implementation, desktop services are treated the same way as mobile services.

Once a service is approved, service manager gains access to the integration properties including AppId, which is then used during the SSO authentication flow.

- AppId — Assigned id used to uniquely identify a registered service.
- PublicKey — The JWT signature public key that can be used to self-verify an access token without delegating the verification to the identity provider.
- FENotificationsEnabled — editable bool property that indicates whether a server uses front end notifications.
- CookieAuthorizationEnabled — editable bool property that indicates whether a service will have refresh token set as an http only cookie (required for services within the same domain) or receive the refresh token together with an access token in a http response.

The identity provider implements the general mechanism mentioned in section 1.6. The SSO session entity is stored in the identity server database and the session cookie is stored on the identity server's /account page which servers as the browser session store. The authentication process is composed of a series of redirects during which the session gets created/updated/deleted. The session entity contains all the currently authenticated services and each time a user requests authentication to another service within the same service group, the session is updated accordingly. When the user requests log out, the session is deleted from the browser and invalidated in the database.

The minimum authentication method feature also utilizes the session entity because the identity server is able to check the current session authentication level based on the already authenticated services.

6.2 Authentication methods

Each service can be configured to use one of the supported authentication methods, ordered by security in ascending order:

- Password — The standard credentials (e-mail, password) method.
- E-mail — User authenticates after validating a code sent to the user's provided e-mail address.
- Mobile application — User authenticates with the KMI Auth mobile application by scanning a QR code presented on the login page.

Each service can be configured to have a minimum authentication method. Methods weaker than the configured minimum method will not be available to the user during authentication (stronger authentication enforcement). For example: if the minimum configured method is password, all methods will be available and presented to the user. If the minimum configured method is e-mail, user will have to authenticate with either e-mail or mobile application.

During the SSO flow, if the user tries to access a service that enforces stronger authentication than all services currently authenticated in the session, they will be required to authenticate again and redirected to the login page with all allowed login methods presented. This mechanism gives service managers control over the service group authentication and enforces the use of a stronger factor while still leaving the choice of the preferred authentication method to the user — unless the service requires mobile authentication only.

Multi-Factor Authentication can be enabled by users to completely eliminate the possibility of a credentials login even if the service group does not enforce it. If enabled, users will always have to authenticate either by e-mail or with KMI Auth.

6.3 Authentication flow

Figure 7 shows a simplified authentication flow. Three actors are present: client service, SSO portal and the identity server. Each step is further described in detail. All identity server's API endpoints are documented in Postman [3]. Step by step integration tutorial as well as a client API documentation is available to admins and user managers in the SSO portal. Mentioned endpoint URLs will be relative to the identity server's domain.

[3] An API tool for building and testing APIs.

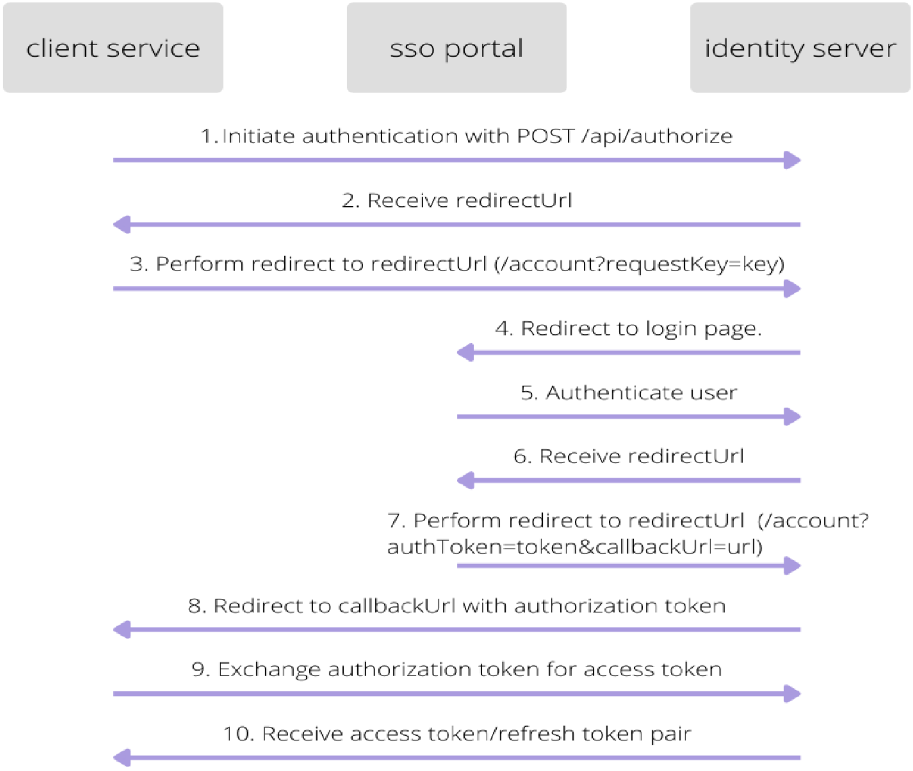

client service
sso portal
identity server
1. Initiate authentication with POST /api/authorize
2. Receive redirectUrl
3. Perform redirect to redirectUrl (/account?requestKey=key)
4. Redirect to login page.
5. Authenticate user
6. Receive redirectUrl
7. Perform redirect to redirectUrl (/account?authToken=token&callbackUrl=url)
8. Redirect to callbackUrl with authorization token
9. Exchange authorization token for access token
10. Receive access token/refresh token pair

Figure 7: SSO flow

1. Authorize

When client service starts initiating authentication, it sends an HTTP POST request to `/api/authorize` endpoint with JSON body:

```
{
   "appId" : "95afe6b8-94b2-498a-924e-edaa36a430f9",
   "authorizeUrl" : "http://clientwebsite.cz/authorize",
   "codeChallenge": "asd78asd_a809das80d_dasdasd8da2" //OAuth PKCE
}
```

Source code 4: Authorize endpoint

AppId is the assigned service id and can be found in the portal's service detail page. AuthorizeUrl is the callback URL where an authorization token will be sent after a successful authentication. The authorizeUrl is also registered with the service and therefore the passed authorizeUrl must match the registered one, otherwise it is considered to be malicious behavior. AuthorizeUrl must be an absolute URL.

The authorize endpoint first verifies the AppId and AuthorizeUrl parameters. Then, a new login attempt is created with a randomly generated request key property. The request key is used to identify the login attempt created by a client service. The code challenge is saved so it can be used later to match with the code verifier. Request key's default expiration time is 15 minutes. Therefore, if authentication does not happen in that time window, a new login attempt has to be created. Furthermore, request key is one-use only.

2. Redirect to identity server

After login attempt has been successfully created, the authorize request finishes by returning a response containing a redirect URL pointing to the identity server's session storage page. The redirect URL is in the form of /account?requestKey=key.

3. Client service redirect

After receiving the redirect URL from the authorize response, the client service performs a redirection to it, delegating the authentication to the identity server. The identity server validates the login attempt. If valid, it looks for an existing browser session by checking the session cookie presence, and matches it with the database session entity. If there is no session cookie present, the user is redirected to the login page where they are requested to authenticate.

If session exists and is valid, an authorization token is created and the identity server redirects to the authorizeURL passed during the initial authorize request with the authorization token attached. For example, the service's authorizeURL is http://client-service.com/authorize. Then, the authorization token will be appended as a query parameter and the resulting URL will be as follows: http://client-service.com/authorize?authToken=token. The session is updated with the newly authorized service added to the session's authenticated services group (considering the service belongs to the same service group).

With every successful authentication, a login stamp is created. The login stamp contains information about the authentication method used, browser user agent, platform and IP address of the user trying to authenticate. The login history can be viewed in the SSO portal.

4. Redirect to login page

The flow demonstrated in Figure 7 considers a nonexistent session. Therefore, the user trying to authenticate will be redirected to the SSO portal's login page where they will be provided with available authentication methods.

5. Authentication & 6. Authentication result

Each authentication method has an individual endpoint for verifying the user in a way specific to the method but the rest of the flow is shared between them, making it easy to add new authentication methods in the future. All authentication requests return a redirectURL pointing to the identity server's session storage page with authenticationToken and callbackURL passed as query parameters: /account?authToken=token&callbackUrl=url.

7. Session creation & 8. Redirect to authorizeURL

Once redirected to the account page, a new session entity is created and the requesting client service is added to a group of services authenticated within the session. An http-only session cookie is set to the identity server's account page. After a session is established, the identity server redirects to the authorizeURL in the same exact way as described in step 3.

9. Authorization token exchange & 10. Token result

Once the client service receives the authorization token passed to the registered authorizeURL, it can be exchanged for a pair of access/refresh tokens using the token endpoint. Both JWT and refresh token contain reference to the session they are associated with.

HTTP POST request to `/api/token` with JSON body:

```
{
    "appId" : "ac04f1f0-337d-4d60-9c80-6e2c5d72a65a",
    "authToken" : "SRO23cZyL0O40T7qVf6JYQM3EJnKDd5...",
    "codeVerifier": "asd78asd_a809das80d_dasdasd8da2" //OAuth PKCE
}
```

Source code 5: Authorize endpoint

Identity server verifies the authToken's and codeVerifier's validity to make sure the service requesting the token exchange is the same service that originally requested authorization. The final token response:

The refresh token's presence in the response is determined by the CookieAuthorizationEnabled service property. If the property is enabled, the refresh token will be set by the server as an http only cookie, otherwise it will be passed together with an access token as showed in the example response.

```
{
    "token": "eyJhbGciOiJIUzI1NiIsInR5..." //jwt access token,
    "expiresIn": "600", //seconds
    "refreshToken": "zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI..."
}
```

Source code 6: Authorize endpoint

6.4 Single Logout

If client service needs a user to logout, the POST logout endpoint /api/logout can be used. It accepts a JSON body containing an optional callbackUrl parameter which indicates a URL where the identity server should redirect once the logout is finished. If the callbackUrl is omitted, the service's registered URL will be used.

When a user initiates logout, all services currently authenticated in the session should be notified about the event and clear all login related state. As mentioned in the OIDC logout section 3.2, the notification system can be implemented as a page with rendered iframes for each service. Since the solution is outdated and the use of iframes is being discouraged due to browsers blocking 3rd party cookies, a different notification approach has been taken.

Server-sent events are a part of the Web API and allow one-directional data flow from server to client (in contrast with two-directional Websocket API data flow). HTTP GET request is used with a text/event-stream response header. The event stream is a stream of UTF-8 encoded text data. Messages are separated by newline characters and are composed of fields formatted as field:value. The Javascript EventSource object is available for SSE subscription functionality. [13]

Example event:

- id:1000 — the event ID
- event:logout — the event name
- data:somedata — the event data (can be stringified JSON)
- retry:10000 — time in ms after which browser will try to reconnect after lost connection to server

The server implementation utilises an asynchronous message queue. Each time a new event needs to be delivered, a message is enqueued and picked up by a listener that dispatches the events to subscribed clients. The process is demonstrated in Figure 8.

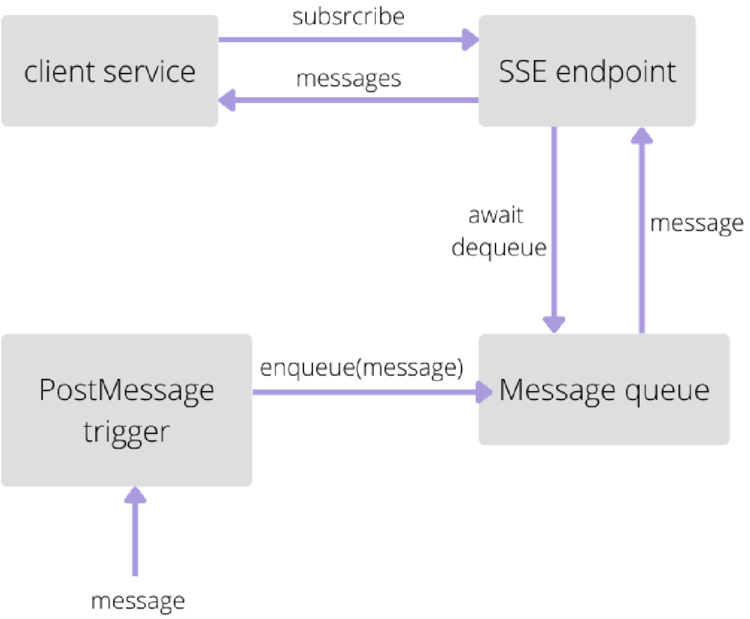


Figure 8: SSE flow

SSE events are targeted only at front end applications. The functionality can be enabled by checking the FE notifications checkbox in service detail in the SSO portal. In the current implementation, only the logout event is available for client services. Clients are required to generate a UUID (subId) value and save it to local storage (or other persistent browser storage). This subscription id combined with the client's appId uniquely identifies the client service in the session and therefore can receive targeted events. To subscribe to notifications, clients can use the notifications endpoint with HTTP GET request to `/api/notifications/subId_appId`.

Once the user is logged out of the session, another problem arises — JWT tokens cannot be revoked. Unless they are expired, JWT tokens remain valid and there is no straightforward mechanism to revoke them. Making the expiry time very short (seconds) is a possibility if the use case permits a few seconds in which the token will still be valid after log out. This will, however, result in a heavier server load because the JWT token will need to be refreshed too often.

To solve this problem, KMI SSO adds a custom claim to the JWT which helps with identifying the session it is associated with. Furthermore, a middleware logic was added to verify the validity of session with every authorization request. The downside to this solution is the lost of efficiency caused by the newly introduced database check.

6.5 SSO Portal

The SSO portal is an interface between the identity server and the user. It enables users to manage their accounts and configure the SSO system as needed. The user interface is rendered depending on the authenticated user's role and is mobile friendly. The portal offers Czech and English language support.

Regarding the question of safely storing access tokens in the browser, a different approach was implemented. Instead of saving the access token to local storage, the access token is not saved at all and is only persisted in memory until the user performs a page refresh. The approach is only possible because of the refresh token mechanism. With the refresh token stored in an http only cookie, each time a page is loaded a new access token is obtained.

As mentioned in the Identity Server section 6.1, the SSO system defines three user roles: user, service manager and admin. User account settings are the same across all roles in the current implementation. Users can change their password, update their profile information (name, surname) and enable MFA. Each role has specific functionality and features available:

User

Apart from account settings, users are only able to browse services they have access to within the SSO system.

Service Manager

Service manager account enables services and user management. Services can be registered, deleted and service information can be updated. The service detail contains integration properties needed for implementing the authentication flow. Every service contains a blacklist of users whose access to the service will be blocked. Services are listed and separated by their registration state to either pending (waiting for admin approval) or registered. Service's login history can be viewed in the service detail. Users can be browsed, added and deleted. Page with documentation is available with a detailed step-by-step tutorial and an API endpoints overview.

Admin

Admins can browse, edit and delete all registered services and system services (SSO portal itself and KMI Auth). Service managers can be deleted and blocked. Service manager deletion has a cascading behavior, ultimately deleting all service group's services and users. A blocked service manager and all users registered in their service group lose access to the SSO. Admin's main responsibility is to approve or reject service manager and service registration requests, which are included in the requests page accessible from the menu. Admin can also view the service manager's documentation.

6.6 KMI Auth

The mobile application serves as an identity verification tool for the SSO system. It is considered to be the strongest authentication method out of the three methods available. The application reuses SSO portal's login page for authentication by opening the login page in a dedicated webview. In order to be able to receive the authentication token callback, a native authorize URL is registered (e.g expo://authorize/token). This is a standard process used by other well known identity providers such as Facebook or Google.

After a successful authentication, the user is presented with a QR scanner and can visit a brief profile screen using a bottom tab menu. After scanning a QR code, the user is presented with a success of failure result screen. Because mobile applications do not use cookies, access token and refresh token should be stored in a secure storage (Keychain in iOS or Keystore in Android). Log out action simply removes locally saved credentials and presents the login screen.

In order for the authentication to be automatic and seamless, the SSO portal needs to be notified about user's successful authentication. SSE can be utilized to push a notification to the SSO portal after the QR code is scanned. Figure 9 visualizes the QR code authentication flow and each step is described.

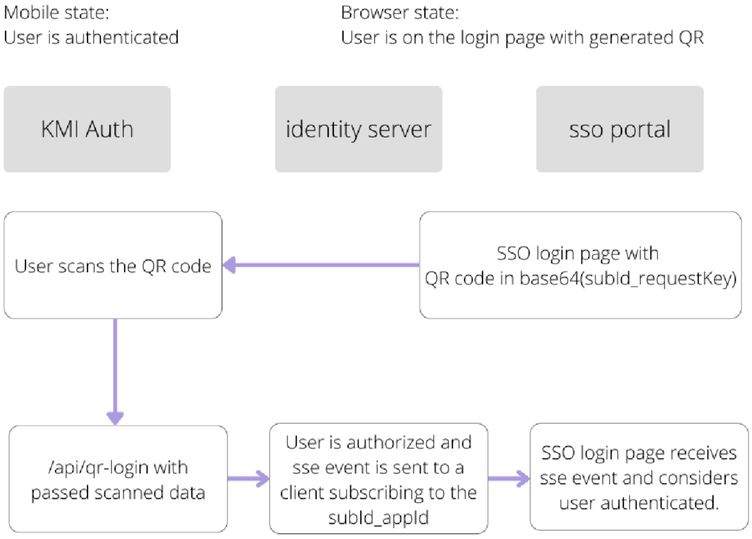


Figure 9: SSE QR implementation flow

1. User is trying to authenticate and chooses the mobile application method. QR code is already generated on the login page. The QR code contains

base64 encoded, underscore-separated data composed of a subId and requestKey.

2. User scans the QR code in the mobile application.

3. Mobile application calls its designated login endpoint POST /api/qr-login and passes the scanned data in the request body.

4. Identity server authorizes the user. If successful, it pushes an SSE login notification to the client subscribing to the /api/notifications/subId_appId endpoint with a redirect URL pointing to the SSO's session storage page attached in the data field.

5. After SSO portal receives the login notification, it performs a redirect and the flow continues in the standard way — session is created/updated and authentication token is sent to the client's authorize URL. User is now authenticated.

Conclusions

Cílem této práce byla implementace systému jednotného přihlášení s důrazem na co největší univerzálnost. V první části práce byl čtenář uveden do problematiky identity, autentizace a autorizace. Dále byla provedena analýza existujících protokolů OpenID Connect a SAML, které umožňují jednotné přihlášení. Jednotlivé protokoly byly popsány a následně srovnány. Větší důraz byl kladen na protokol OpenID Connect, jelikož slouží jako teoretický základ a inspirace k následnému vlastnímu řešení. Porovnány byly také populární existující systémy Okta, Auth0 a Keycloak, umožňující jednotné přihlášení.

Druhá část práce se zaměřuje na popis návrhu implementace jednotného systému přihlášení. Byly popsány jednotlivé aplikace, ze kterých se systém skládá — server spravující identity uživatelů, webový portál umožňující správu uživatelů a služeb a mobilní aplikace určená k verifikaci uživatele přes QR kód. Práce se ve větší míře soustřeďuje na klíčové koncepty spíše než technické detaily. U jednotlivých částí je proveden rozbor hlavní funkcionality a dostupných funkcí.

Výsledkem práce je ekosystém služeb jednotného systému přihlášení spolu s dokumentací implementace a serverového API, určeného především pro integraci delegace autentizace klientskými službami. Výsledné programy, mimo jiné, mohou sloužit jako ukázková implementace a důležité části zprostředkovávající komunikaci mezi klientskou službou a identity serverem mohou být přepoužity jako knihovny.

Conclusions

The aim of this thesis was the implementation of a single sign-on system with an emphasis on its versatility. In the first part of the thesis, the reader was introduced to the issue of identity, authentization and authorization. Moreover, the thesis analysed two existing protocols, OpenID Connect and SAML, which enable single sign -on. The OpenID Connect protocol was the centerpiece of the thesis since it serves as the theoretical basis and inspiration for the subsequent custom solution. Finally, three popular exiting systems facilitating single sign-on (Okta, Auth0 and Keycloak) were compared.

The second part of the thesis describes the design and implementation of a single sign-on system. The individual applications constituting the system were described — an identity server that provides user identity management, a web portal providing user and service management, and a mobile application that enables verification via a QR code. The thesis is more focused on key concepts rather than technical details. For each part of the system, there is an analysis of the main functionality and available features.

The result of the thesis is a single sign-on ecosystem with documentation and implementation of a server API used mainly for integrating a client service authentization delegation. Moreover, the final programs can be used as an example implementation, and the important sections mediating communication between the client service and the identity server can be reused as libraries.

A Contents of attached CD/DVD

All setup instructions are located in the README file. The project is fully dockerized and it is recommended to use docker-compose.

`backend/`
Complete folder structure and source codes of the identity server ASP.NET Core solution. The identity solution itself is composed of four projects: Api, Application, Infrastructure and Domain. Furthermore, two example projects are present: Client1 and Client2.

`frontend/`
Complete folder structure and source codes of the SSO portal NextJS application.

`mobile/`
Complete folder structure and source codes of the KMI Auth React Native application.

`README.md`
The readme file contains detailed installation instructions, available testing user accounts with passwords and links to external API and database documentations.

`docker-compose.yaml`
The docker compose file that creates and starts all project containers.

`init.sh`
Initialization script used to replace project URL addresses with user's local IP address. This is necessary for testing of the mobile application on localhost.

`doc/`
The doc folder contains all created diagrams used in this thesis and the user interface design of the SSO portal (Sketch). The thesis text is also located in this folder.

`api-templates/`
This folder contains an API template used by the OpenAPI tool to generate an API client in Typescript.

`RSA/`
Complete folder structure and source codes of a simple .NET Core RSA generator utility.

All materials attached on CD/DVD that have been borrowed are either not subject to copyright or their further distribution is allowed by the author. All (cited) materials, for which this statement is not true, therefore they are not included on the CD/DVD, have their source attached in the bibliography, the text itself or in the file README.md

Glossary

2FA Two factor authentication

CORS Cross-origin resource sharing

JWE JSON Web Encryption

JWS JSON Web Signature

JWT JSON Web Token

MFA Two factor authentication

OIDC OpenID Connect

PKCE Proof Key for Code Exchange

SAML Security Assertion Markup Language

SPA Single page application

SSE Server-sent events

SSO Single sign-on

UUID Universal Unique Identifier

XSS Cross-site scripting

References

[1] Yvonne Wilson, Abhishek Hingnikar. *Solving Identity Management In Modern Applications - Demystifying OAuth 2.0, OpenID Connect, And SAML 2.0*. First. San Francisco: Apress, 2019. xvii, 323 pp. ISBN 978-1-4842-5094-5.

[2] Hauk, Chris. Browser Fingerprinting: What Is It And What Should You Do About It?: Browser fingerprinting identifies you and collects data about your online travels. In this article, we'll discuss how browser fingerprinting works and how you can prevent it. [online]. 2022, [visited on 2022-6-22]. Available from WWW: ⟨https://pixelprivacy.com/resources/browser-fingerprinting/⟩.

[3] Coursicle.com. Loginless – A New Standard for User Identification. [online]. 2022, [visited on 2022-6-22]. Available from WWW: ⟨https://www.coursicle.com/blog/loginless-a-new-standard-for-user-identification.php⟩.

[4] Hardt, D. The OAuth 2.0 Authorization Framework. [online]. 2012, [visited on 2022-6-28]. Available from WWW: ⟨https://datatracker.ietf.org/doc/html/rfc6749⟩.

[5] N. Sakimura, Microsoft. Proof Key for Code Exchange by OAuth Public Clients. [online]. 2015, [visited on 2022-6-28]. Available from WWW: ⟨https://datatracker.ietf.org/doc/html/rfc7636⟩.

[6] Justin Schuh, Google. Building a more private web: A path towards making third party cookies obsolete. [online]. 2020, [visited on 2022-6-30]. Available from WWW: ⟨https://blog.chromium.org/2020/01/building-more-private-web-path-towards.html⟩.

[7] N. Sakimura J. Bradley, Microsoft. OpenID Connect Core 1.0. [online]. 2014, [visited on 2022-7-1]. Available from WWW: ⟨https://openid.net/specs/openid-connect-core-1_0.html⟩.

[8] M. Jones J. Bradley, Microsoft. OpenID Connect Back-Channel Logout 1.0 - draft 08. [online]. 2022, [visited on 2022-7-10]. Available from WWW: ⟨https://openid.net/specs/openid-connect-backchannel-1_0.html⟩.

[9] M. Jones, Microsoft. OpenID Connect Front-Channel Logout 1.0 - draft 06. [online]. 2022, [visited on 2022-7-10]. Available from WWW: ⟨https://openid.net/specs/openid-connect-frontchannel-1_0.html⟩.

[10] M. Jones, Microsoft. JSON Web Token (JWT). [online]. 2015, [visited on 2022-7-1]. Available from WWW: ⟨https://datatracker.ietf.org/doc/html/rfc7519⟩.

[11] Nick Ragouzis John Hughes, Rob Philpott et al. Security Assertion Markup Language (SAML) V2.0. [online]. 2008, [visited on 2022-7-5]. Available from WWW: ⟨https://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html⟩.

[12] Okta. SWA app integrations. [online]. 2022, [visited on 2022-7-6]. Available from WWW: ⟨https://help.okta.com/en-us/Content/Topics/Apps/apps-about-swa.htm⟩.

[13] contributors, MDN. Using server-sent events. [online]. 2022, [visited on 2022-7-10]. Available from WWW: ⟨https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events⟩.