Information Technology Department
Collaboration, Devices and Applications - Integrated Collaboration Group
(IT-CDA-IC)

MALT

Reproducible Examples for Integration with Keycloak

Supervisors: Hannah Short, Paolo Tedesco
Mohammad El Hajj Hussein

Summer Student Program 2019
Acknowledgement

I would like to thank my supervisors Hannah Short and Paolo Tedesco, whose direction, guidance, and insight were essential for the success of this project. I would like to also thank my teammates and office co-workers who generously provided me with their time and vast knowledge. I am very thankful for the IT secretariat and all the people who gave me a chance to have this wonderful experience. And of course, the support of my wife, Batoul Ghaddaf, who’s motivation was essential for the success of this project.
1 Abstract

The Microservice architecture on Libre Technology (MALT), “Microsoft Alternatives” previously, Project started a year ago to mitigate anticipated software license fee increases. CERN has been working on a new Authentication and Authorisation system, based on open source products that enable modern protocols (OIDC, OAuth2, SAML) and improve the user experience. CERN services will all need to be enhanced to interact with the new Authentication and Authorisation Infrastructure (an open source software stack using Keycloak for Authentication). This project involves developing reproducible examples to demonstrate the necessary configuration to service developers, along with high quality documentation. These examples are written using different languages in order to cover different types of applications. These applications include: Front-end based authentication Web application (written in JavaScript), Back-end based authentication Web application (one written in JavaScript and other written in Python), and Mobile Native application (Android application written in Java).

2 Introduction

2.1 MALT

Over the years, CERN’s activities and services have increasingly relied on commercial software and solutions to deliver core functionalities, often leveraged by advantageous financial conditions based on the recognition of CERN’s status as an academic, non-profit or research institute. Once installed, well-spread and heavily used, the leverage used to attract CERN service managers to the commercial solutions tends to disappear and is replaced by licensing schemes and business models tuned for the private sector. Given the collaborative nature of CERN and its wide community, a high number of licenses are required to deliver services to everyone, and when traditional business models on a per-user basis are applied, the costs per product can be huge and become unaffordable in the long term.

A prime example is that CERN has enjoyed special conditions for the use of Microsoft products for the last 20 years, by virtue of its status as an “academic institution”. However, recently, the company has decided to revoke CERN’s academic status, a measure that took effect at the end of the previous contract in March 2019, replaced by a new contract based on user numbers, increasing the license costs by more than a factor of ten. Although CERN has negotiated a ramp-up profile over ten years to give the necessary time to adapt, such costs are not sustainable.

Anticipating this situation, the IT department created the “MALT” project a year ago. The initial objective was to investigate the migration from commercial software products (Microsoft and others) to open-source solutions, to minimise CERN’s exposure to the risks of unsustainable commercial conditions.
2.2 Authentication of User

Authentication is about validating your credentials like User Name/User ID and password to verify your identity. The system determines whether you are what you say you are using your credentials. In public and private networks, the system authenticates the user identity via login passwords. Authentication is usually done by a username and password, and sometimes in conjunction with factors of authentication, which refers to the various ways to be authenticated.

2.3 OIDC

OpenID Connect is an interoperable authentication protocol based on the OAuth 2.0 family of specifications. It uses straightforward REST/JSON message flows with a design goal of “making simple things simple and complicated things possible”. It’s uniquely easy for developers to integrate, compared to any preceding Identity protocol.

OpenID Connect lets developers authenticate their users across websites and apps without having to own and manage password files. For the app builder, it provides a secure verifiable answer to the question: “What is the identity of the person currently using the browser or native app that is connected to me?
3 Keycloak

Keycloak is an open source Identity and Access Management solution aimed at modern applications and services. It makes it easy to secure applications and services with little to no code. Keycloak is based on standard protocols and provides support for OpenID Connect, OAuth 2.0, and SAML.

Users authenticate with Keycloak rather than individual applications. This means that your applications do not have to deal with login forms, authenticating users, and storing users. Once logged in to Keycloak, users do not have to log in again to access a different application. This is also applied to logging out. Keycloak provides a single sign out, which means users only have to log out once to be logged-out of all applications that use Keycloak.

The default theme of the Keycloak log-in page can be changed. It is temporarily as shown in figure 1, however it will be changed to a fancier one later on.

Figure 1: CERN Log-in page theme
3.1 Registering Your Application

During the process of authentication, the client (application) needs to send its ID and secret. These credentials are obtained by registering the client in the authentication server. However, in order to ensure that no wrong configurations happen during the registration of the application, CERN developers produced an application portal that provides a link between the authentication server and the client owner (figure 2).

![Application Portal](https://application-portal.web.cern.ch/)

After registering the application, an ID and secret are provided that uniquely identify this application.

3.2 Types of Flow

There are three types of Authentication Flows:

3.2.1 Code Flow:

With this flow the Keycloak server returns an authorization code, not an authentication token, to the application. The client exchanges the code for an access token and a refresh token after the browser is redirected back to the application.

3.2.2 Implicit Flow:

Keycloak also supports the *Implicit flow* where an access token is sent immediately after successful authentication with Keycloak. This may have a better performance than standard flow, as there is no additional request to exchange the code for tokens, but it has implications when the access token expires. However, sending the access token in the URL fragment can be a security vulnerability. For example, the token could be leaked through web server logs and/or browser history.
3.2.3 Hybrid Flow:

The Keycloak server will then send both the code and tokens to your application. The access token can be used immediately while the code can be exchanged for access and refresh tokens. Similar to the implicit flow, the hybrid flow is good for performance because the access token is available immediately. But the token is still sent in the URL, and the security vulnerability mentioned earlier may still apply. One advantage in the hybrid flow is that the refresh token is made available to the application.

4 Applications Developed

The main goal of this project is to create examples that illustrate the necessary configuration to service developers in order to integrate the client (new application) with the Keycloak. Being an open source program, Keycloak does not have vivid details for integration with different languages. This also reaches the level where no Keycloak libraries or no illustrated examples are provided. All examples developed in this project can be found [here](#).
4.1 Front-end based authentication Web application

The example is implemented in JavaScript. Keycloak provides an organized adapter that manages the authentication of the user. After importing the Keycloak adapter, you need to configure Keycloak with the authentication URL, the real name, and the client ID that is registered in the authentication server (figure 4a). Then, the authentication is initialized by choosing the “implicit flow” since the authentication takes place in front-end, so you don’t want to leak the client secret (figure 4b). After successful authentication, the encoded and decoded tokens are displayed (figure 4c).

```javascript
var keycloak = Keycloak({
  url: 'https://keycloak.cern.ch/auth',
  realm: 'XXXXXX',
  clientId: 'XXXXXX'
});

keycloak.init({ onLoad: 'login-required', flow: 'implicit' }).success(function(authenticated) {
  if(authenticated) {
    displayTokenInfo(keycloak);
  } else {
    error('failed to initialize');
  }
});

const displayTokenInfo = (keycloak) => {
  const rawToken = document.getElementById('raw-token');
  rawToken.innerHTML = '<p>j(' + keycloak.token + ')</p>
  const parsed = document.getElementById('parsed-token');
  parsed.innerHTML = '<pre>{' + prettyToken + '</pre>;
  const prettyToken = JSON.stringify(keycloak.tokenParsed, null, '\t');
      }

  (a) Keycloak configuration

(b) Initializing Authentication

(c) Presenting Tokens

Figure 4: Authentication example using JavaScript

4.2 Back-end based authentication Web application

Since the authentication here takes place at the server level, the client secret can be stored safely at the server. In addition to that, “Authentication Code Flow” is used (i.e. token exchange takes place).

4.2.1 Golang-based

No Keycloak adapter exists for the Go-Lang based applications. Thus, I needed to use a more basic library (go-oidc library). The configurations are set (figure 5a). The user’s state is read
from and stored in the session. Upon starting, the user’s state is checked to see if the user is logged in. If not, the user is redirected to the authentication server (figure 5b). After logging in, the request is decoded, and the tokens are displayed (figure 5c).

```go
oauth2Config := oauth2.Config{
    ClientID:  clientID,
    ClientSecret: clientSecret,
    RedirectURL: redirectURL,
    // Discovery returns the OAuth2 endpoints.
    Endpoint: provider.Endpoint(),
    // "openid" is a required scope for OpenID Connect flows.
    Scopes: []string{oidc.ScopeOpenID, "profile", "email"},
}
```

(a) Keycloak configuration

```go
oauthState := uuid.New().String()
.sess.Set(state, oauthState)

// checking the userinfo in the session.
// If it is nil, then the user is not authenticated yet
userInfo := sess.Get("userinfo")
if userInfo == nil {
    return
}
```

(b) checking if logged in. If not, the user is redirected to authentication server

```go
resp := struct {
OAuth2Token *oauth2.Token
IDTokenClaims *json.RawMessage // ID Token payload is just JSON.
}{oauth2Token, new(json.RawMessage)}
err = idToken.Claims(&resp.IDTokenClaims)
```

(c) Reading the response

Figure 5: Authentication example using GoLang

### 4.2.2 Python-based

The same concept of integration is applied here. Also, the Keycloak configurations are now read from a json file (figure 6a). However, there exists a Keycloak adapter, but still primitive, that makes the authentication process - initializing (figure 6b) and exchanging tokens (figure 6c) - smoother.
4.3 Android-based Native Mobile application

For the Android applications, Keycloak recommends the use of another open source library “AppAuth”. Unlike the web applications where the authentication state is saved in the session of the browser, the authentication state here is saved in the form of “shared preferences”. In order to manage the state, we created the class “AuthStateManager”. This class manages the process of updating and reading the authentication state by controlling the “shared preferences” file to make it consistent in the application. The configurations are set whenever there is a need for them.

However, another class that manages the configurations can be implemented. Upon start, the user is asked to sign in (figure 7a). After signing in through Keycloak, the user is redirected to the main page of the application. In our application, it is just a page that offers an API call and signing out (figure 7b). By choosing to make an API call, authentication code flow will take place. Then user’s tokens and credentials are displayed (figure 7c). By choosing to sign out, the ‘log out’ endpoint of Keycloak is contacted, and the authentication state is cleared.
4.4 Desktop Native Application

We were trying to develop a Desktop Native app using Electron library. However, Keycloak provides a NodeJS library for authentication but not specifically for desktop native apps. I tried to manage the authentication request by writing the functions from scratch.

The app was working and it was sending the request with the ID and secret of the application. The user is able to submit his credentials in order to be verified. However, The response that the Electron App is getting from the Keycloak server is null. I couldn’t fix this error.
5 Device Flow

The Device Grant (formerly known as the Device Flow) is an OAuth 2.0 extension that enables devices with no browser or limited input capability to obtain an access token. This is commonly seen on Apple TV apps, or devices like hardware encoders that can stream video to a YouTube channel.

5.1 The Flow Process

The first step in the process is for the client device to ask our authorization server for access. In return, our authorization server responds with: a device code, a user code, and a verification URI. The device will then transmit to the user, the user code, and verification URI, asking the user to visit this URI and enter the code.

When the user visits this site, the authorization server needs to authenticate the user (if they haven’t already done so, hooray for SSO). Once they have verified their identity, they enter the user code and give their consent to the client device. It is the user’s responsibility to confirm that the client application requesting authorization matches the one that is making the request.

While this is happening, the device is polling the authorization server’s token endpoint, with its device code, asking: “Have they authorized me yet? Have they authorized me yet?”. This goes on until the authorization server says yes or gets annoyed enough to turn the car around. Upon authorization, the authorization server returns the tokens in response to the polling.

5.2 Device Flow with Keycloak

We wanted to develop a device flow PAM module example using QR-codes that the user can scan to get the link. Keycloak documentation doesn’t tackle in anyway this flow. Unfortunately, after further research, it turns out that Keycloak doesn’t support the device flow. Therefore, we were forced not to continue this example.
6 Conclusions

In conclusion, as part of the MALT project, the transition to the new Authentication and Authorization System (Keycloak) is essential. Being an open source program, Keycloak doesn’t have sufficient documentation and libraries to rely on for the process of securing your applications. This project focused on demonstrating the process of integrating different types of applications whilst providing a detailed README file. This can facilitate the job of service developers.

One can make contribution to the community by continuing this work. More documented and smooth coded libraries can be implemented. Contributions should also be made where there is a basic need for it. For example, Keycloak official website doesn’t comment about desktop native applications (developed by Electron as an example).
References

[1] https://malt.web.cern.ch/malt/project/
