Cahier de TP
« Gestion centralisée de la sécurité avec KeyCloak»

Pré-requis :

- Bonne connexion Internet
- NodeJS, npm
- Docker / JDK17
- Git
- IDE Java
- Maven ou Quarkus CLI

Les chemins indiqués dans les énoncés de TP sont relatifs à la racine du dépôt git

Certaines solutions sont disponibles à :
https://github.com/dthibau/keycloak-solutions

Table des matières

Atelier 1: Installation et sécurisation d'une première application

1.1 Installation

JDK17

Page de téléchargement : ***https://www.keycloak.org/downloads***

L'extraire dans un répertoire de votre choix

Démarrer en mode développement

Linux, Git bash :

```
bin/kc.sh start-dev
```

Windows CMD, PowerShell

```
bin/kc.bat start-dev
```

Accéder à http://localhost:8080 et créer un utilisateur admin : ***admin/admin***

Docker

Utiliser le docker compose fourni

```
cd 1_FirstAppli
docker compose up -d
```

Cela démarre 3 services :

- Serveur Keycloak
- Base Postgres
- Client pgAdmin

1.2. Création d'un realm

Accéder à la console d'administration

Créer un realm nommé ***formation***

Pour le realm *formation* :

- Créer un user avec comme login ***user***
- Dans l'onglet ***Credentials,*** définir le mot de passe ***secret*** et décocher la check-box ***temporaire***
- Créer un rôle REALM : ***ROLE_USER***

- Créer un groupe ***GROUP_USER*** et y affecter l'utilisateur précédent, lui affecter également le rôle

Se connecter avec *user/secret* à la console des comptes utilisateur (*http://localhost:8080/realms/formation/account*), et parcourir l'interface.

1.3. Sécurisation d'une application

1.3.1 Démarrage des applications

Récupérer les sources fournis

Démarrer l'application frontend :

```
cd frontend
npm install
npm start
```

Puis dans une autre fenêtre, l'application back-end

```
cd backend
npm install
npm start
```

Accéder à *http://localhost:8000* et vérifier la présence d'un bouton login

1.3.2 Enregistrement du client frontend

Dans la console d'admin de Keycloak, créer un nouveau client :

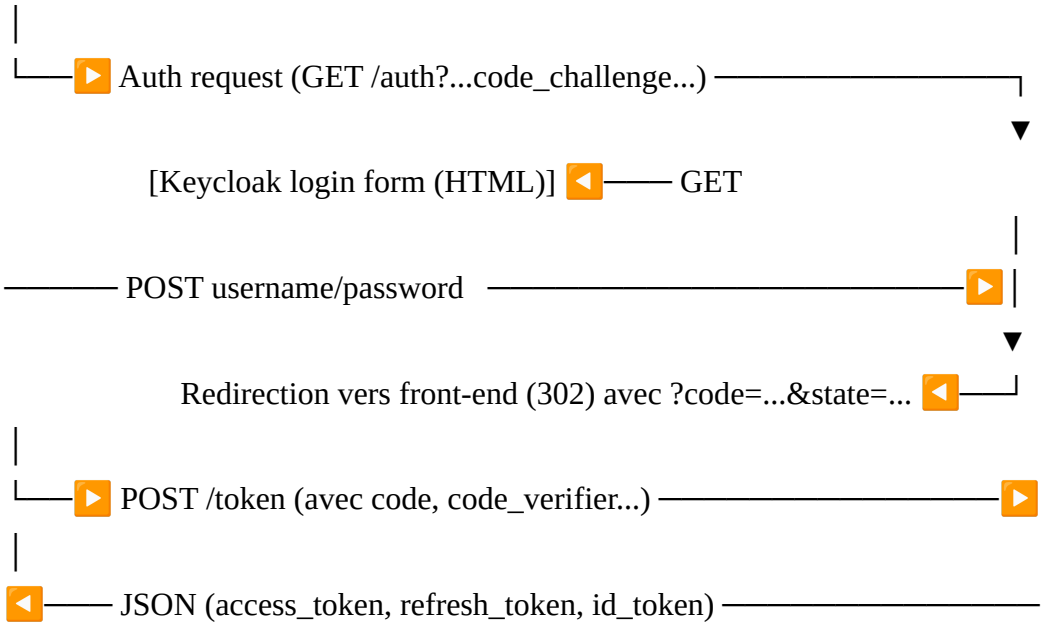
- Client Type : ***OpenID***
- Client ID: ***frontend***
- Client authentication : ***Off***
- Authorization : ***Off***
- Authentication Flow : ***Standard Flow (Authorization Code)***
- Décocher : ***Direct Access Grant (Password Grant Type)***

URLs

- Root URL: ***http://localhost:8000***
- Valid Redirect URI : ***http://localhost:8000/***
- Logout URL : + (cela inclut toutes les *Valid Redirect URIs*)
- Web origins : +

Activer ensuite le bouton ***login*** dans l'application *frontend*, vous devriez vous logger sur Keycloak et accéder aux boutons permettant de voir les jetons obtenus.
Vous pouvez activer la console de debug du navigateur sur l'onglet réseau et ne filtrer que les requêtes HTML et XHR.

Essayer de visualiser les URLs suivantes :
[Frontend SPA – login]



Visualiser les jetons fournis par Keycloak, vérifier que l'utilisateur a bien le rôle ***ROLE_USER***

1.3.3 Attribut personnalisé et accès au backend

Vous pouvez également renseigner l'attribut ***picture*** de l'utilisateur avec une URL pointant sur une image afin de voir comment une application peut utiliser les attributs stockés dans *KeyCloak*

Accéder directement aux URLs du backend :

- *http://localhost:3000/public*
- *http://localhost:3000/secured*

Accéder ensuite à l'URL sécurisé via l'application de *frontend*, visualiser les traces de l'application backend

Atelier 2: OpenID Connect

Récupérer les sources fournis

Démarrer l'application :

```
cd 2_OpenIDConnect
npm install
npm start
```

Accéder à http://localhost:8000

2.1 Endpoint de découverte

Activer le bouton « *1 – Discovery* »

Vérifier l'URL de l'issuer (il contient le nom du realm Keycloak)

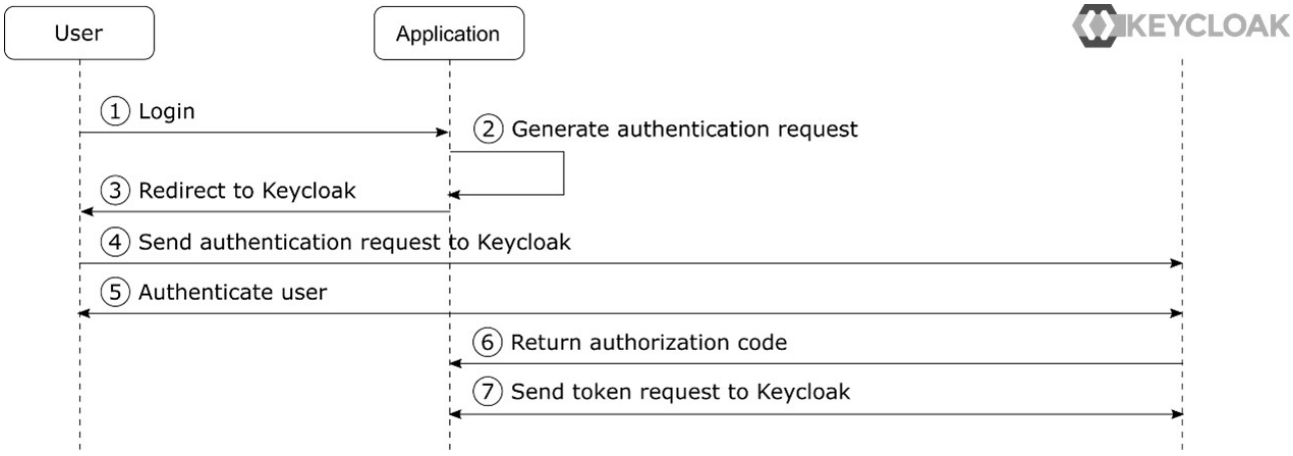
Activer ensuite le bouton « *Load OpenID Provider Configuration* »

Elle provoque la requête GET :
http://localhost:8080/realms/<realm-name>/.well-known/openid-configuration

Visualiser les informations accessibles.

2.2 Authentification

Le processus détaillé de l'authentification est le suivant :



2.2.1. Requête d'authentification

Pour envoyer la requête d'authentification, cliquer sur le bouton correspondant.

Un formulaire est proposée par l'application :

- *client_id* : Le client enregistré chez Keycloak

- *scope* : La valeur par défaut est ***openid*** , ce qui signifie une requête OpenID
- *prompt* : Ce champ peut être utilisé pour différents buts :
  - *none* : Keycloak n'affiche pas d'écran de login mais authentifie l'utilisateur seulement si l'utilisateur est déjà loggé
  - *login* : Demande à l'utilisateur de s'authentifier sur Keycloak même si il est déjà loggé
- *max_age* : Maximum en seconds de la dernière authentification valide
- *login_hint* : Si l'application connaît la login de l'utilisateur, il peut pré-remplir le champ login de Keycloak

Générer la requête et l'envoyer. Le point d'accès est
GET http://localhost:8080/realms/<realm-name>/protocol/openid-connect/auth

La réponse doit contenir le code d'autorisation

2.2.2 Obtention des jetons

La 3ème étape consiste à obtenir les jetons avec le code d'autorisation.

Utiliser les boutons adéquats

Observer la réponse et ses différents champs

Observer le jeton d'identification déchiffré.

2.3 Authentification utilisateur avec certificat X509

Revisitez le docker compose fourni, en particulier :

- La configuration du certificat serveur dans un keystore
- L'activation de l'authenticator X509
- Les options de la JVM donnant accès à un truststore
- L'option HTTPS_CLIENT_AUTH=request
- Les traces pour l'authentification

2.3.1 Création de l'utilisateur alice

Dans le realm formation, créer un utilisateur ***alice*** sans définir de mot de passe

2.3.2 Modification du Browser-flow

Dans le menu ***Authentication***, sélectionner le flow ***browser*** et dupliquer le dans un flow nommé ***browser X509***

Ajouter une étape alternative ***X509/Validate Username Form*** au même niveau que ***Cookie***

Configurer là comme suit :

- Alias : ***Use CN***
- User Identity Source : ***Match subject DN using regular expression***
- Regular expression : ***CN=(.*?)(?:,|$)***
- User Mapping Method : ***Username or email***

Dans la liste déroulante Action, sélectionner ***Bind Flow*** et associer ce flow au ***Browser Flow***

2.3.3 Test de la configuration

Importer le certificat client ***user.p12*** dans votre navigateur. :

Mot de passe : ***plb***

CA : ***PLB CA***

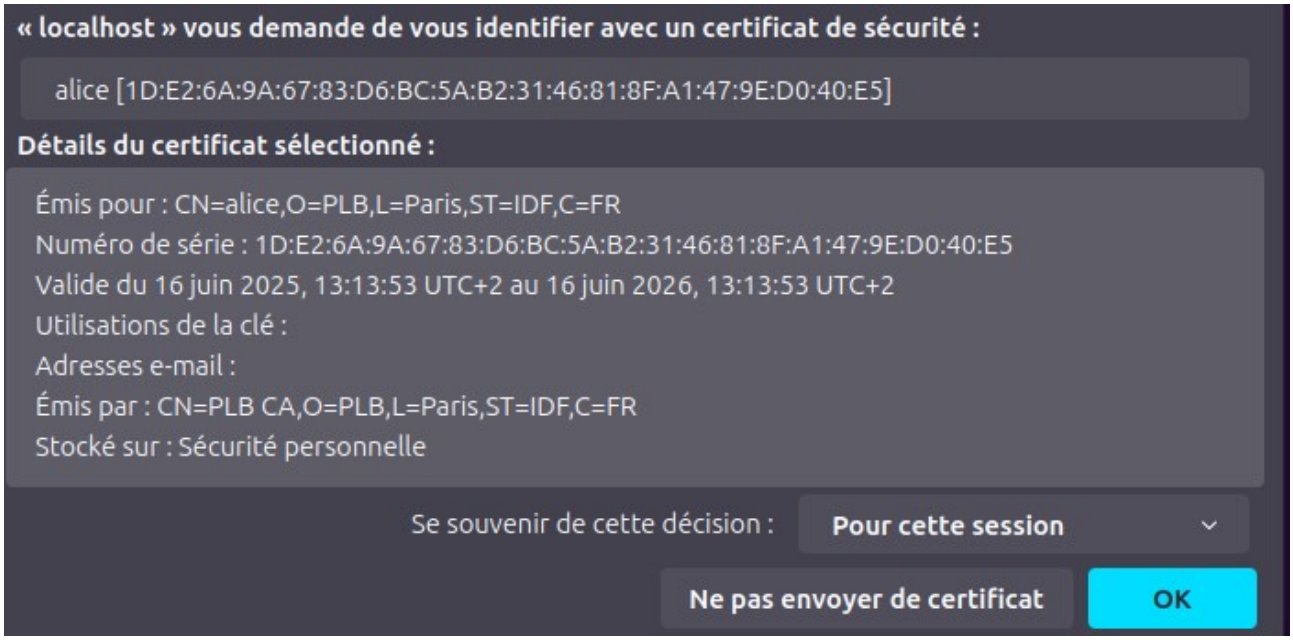
Par exemple, dans firefox :

Paramètres → Certificats → Afficher les certificats → Vos certificats → Importer

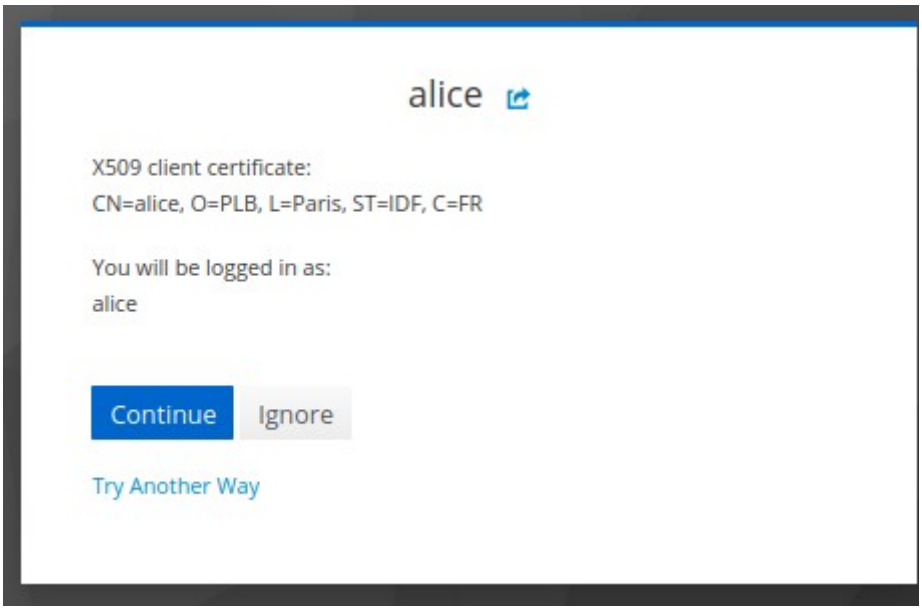
Accéder ensuite à :

https://localhost:8443/realms/formation/account

Vous devez voir la fenêtre suivante :



Envoyer le certificat et activer le bouton Sign in, vous devez voir la fenêtre suivante :



Et vous logger directement

2.4 CIBA Flow

2.4.1 Mise en place realm

Visualiser le docker-compose fourni, en particulier la commande de démarrage.

Démarrer le docker compose :

```
docker compose up -d
```

Accéder à la console d'administration et :

- Créer un relm ***formation***
- Créer un client confidentiel ***ciba-client*** en activant le flow CIBA, noter son crédentiel
- Créer un utilisateur ***user*** sans mot de passe

2.4.2 Application backend

Démarrer l'application backend :

```
cd ciba-backend
./mvnw clean package
java -jar target/ciba-backend-0.0.1-SNAPSHOT.jar
```

Le service démarre sur le port 8082

2.4.3 Test du flow

Effectuer la requête de demande d'authentification auprès de Keycloak :

```
curl -u 'ciba-client:<client-secret>' \
  -d "scope=openid" \
  -d "login_hint=user" \
  -d "binding_message=TestCIBA" \
  http://localhost:8080/realms/formation/protocol/openid-connect/ext/ciba/auth
```

Noter le jeton de la réponse et Visualiser les logs de Keycloak et du backend.

Effectuer la requête de notification du résultat :

```
curl -X POST "http://localhost:8082/confirm?login_hint=user&status=SUCCEED"
```

Visualiser les logs du backend.

Finalement, effectuer la requête récupérant le jeton :

```
curl -X POST
http://localhost:8080/realms/formation/protocol/openid-connect/token \
  -u 'ciba-client:<client-secret>' \
  -d 'grant_type=urn:openid:params:grant-type:ciba' \
  -d 'auth_req_id=<token>'
```

2.5 Création de scope et personnalisation des revendications

Reprendre la première stack avec la base Postgres.

Ajouter un attribut ***no-secu*** à l'utilsateur ***user***

Créer un client scope : ***secu_scope***

Dans l'onglet *Mappers*, créer un mapper :

- Type : ***User Attribute***
- Name : ***no-secu-mapper***
- User Attribute : ***no-secu***
- Token Claim Name: ***no-secu***
- Claim JSON Type: ***String***

S'assurer que cet attribut sera ajouté au jeton d'identification.

Sélectionner ensuite le client *frontend* et ajouter *secu_scope* comme scope optionnel du client

Reprendre l'application node du TP sur l'authorization flow ou utiliser la fonctionnalité *ClientDetail – Client Scopes - Evaluate* de Keycloak

S'authentifier à nouveau en précisant dans le paramètre *scope* : ***openid secu_scope***

Visualiser le jeton d'identification, l'attribut *no-secu* doit être présent

Ajouter un rôle mapping à *secu_scope* (Onglet Scope), affecter le rôle *ROLE_USER*

Créer un utilisateur user ***user_with_no_roles*** sans lui donner le rôle *ROLE_USER*

S'authentifier via l'application avec cet utilisateur et vérifier que l'attribut ***no-secu*** n'apparaît pas.

Accéder ensuite au points d'accès de *UserInfo*.

Sélectionner le client *frontend*, dans client scopes, sélectionner ***frontend-dedicated*** et y créer un nouveau Mapper avec le mapper prédéfini ***Group Membership***

Vérifier

2.6 Back-channel Logout

Pour le client frontend, désactiver le *front-channel* logout

Configurer une URI de Back-channel logout à : http://host.docker.internal:8000/logout

Se logger sur l'application ***frontend***

Avec un autre navigateur, connecter vous en tant qu'admin sur le serveur Keycloak.

Accéder à *Realm Formation → sessions*

Forcer un logout SSO via l'administration

Observer les traces de Keycloak ainsi que celles de l'application *frontend*

Atelier 3 : Jetons d’accès oAuth

Récupérer les sources fournis

Démarrer l’application front-end :

```
cd frontend
npm install
npm start
```

Puis l’application back-end

```
cd backend
npm install
npm start
```

Accéder à *localhost:8000*, s’authentifier puis invoquer le service backend.

3.1 Consentement de l’utilisateur

Avec Keycloak, les applications peuvent être configurées pour exiger ou ne pas exiger le consentement.

Dans la console d’administration, sélectionner le client *frontend* et activer le consentement de l’utilisateur.

Obtenir à nouveau un jeton d’accès via *localhost:8000*

Créer un nouveau scope avec les informations suivantes :

- Name: ***albums***
- Display On Consent Screen: ***ON***
- Consent Screen Text: ***Visualiser votre album photo***

Configurer le client *frontend* pour que ce *scope* soit optionnel.

Obtenir à nouveau un jeton d’accès en précisant le scope albums.

3.2 Limitation des audiences du jeton

Éditer le fichier de configuration du backend *keycloak.json* et passer l’attribut ***verify-token-audience*** à ***true***

Le nom de la ressource est ***backend***

Obtenir à nouveau un jeton d’accès et observer le champ ***aud***

Essayer d’invoquer le service

Dans la console d'admin, créer un client ***backend*** en décochant tous les flows de login

Revenir sur le client *frontend* puis le client scope *frontend-dedicated* et créer un nouveau Mapper de type **audience**:

- Name: ***backend audience***
- Mapper Type: ***Audience***
- Included Client Audience: ***backend***

Essayer d'invoquer le service

3.3 Limitation des rôles du jeton

Rôle realm

Visualiser le jeton d'accès avec la configuration par défaut. Des informations sur les rôles des utilisateurs et les rôles des clients doivent apparaître

Éditer le scope dédié à l'application frontend, accéder à l'onglet *scope* et désactiver « ***Full scope allowed*** »

Visualiser le jeton obtenu avec cette configuration

Retourner dans la configuration du scope dédié et lui affecter le rôle *ROLE_USER*

Visualiser le jeton obtenu avec cette configuration

Retourner dans la configuration du scope dédié et supprimer le rôle *ROLE_USER*

Affecter le rôle USER au client scope *custom_scope* créé au TP précédent

Demander un nouveau jeton en indiquant le scope ***custom_scope***

Rôle client

Ajouter également le rôle ***admin*** au client frontend

Editer le user user et lui affecter le rôle ***front-end:admin***

Visualiser le jeton obtenu

3.4 Limitation des scopes du jeton

Créer 3 scopes :

- ***albums:view***
- ***albums:create***
- ***albums:delete***

Configurer ensuite le client *frontend* afin :

- *album:view* soit un scope par ***défaut***
- *albums:create* et *albums:delete* sont des scopes ***optionnels***

S'assurer que le consentement de l'utilisateur est requis pour le client *frontend*

Demander un nouveau jeton sans indiquer de scope. Le consentement pour le scope *album:view* devrait être demandé

Demander ensuite un nouveau jeton en indiquant un scope optionnel.

3.5 Validation du jeton via le endpoint d'introspection

S'assurer que le client backend a la case cochée « *Client authentication* ».
Récupérer le secret du client backend.

Dans un terminal :

```
export SECRET=….
```

Récupérer le jeton d'accès de l'application front-end, puis

```
export TOKEN=…
```

Valider le jeton via :

```
curl --data "client_id=backend&client_secret=$SECRET&token=$TOKEN"
http://localhost:8080/realms/formation/protocol/openid-connect/tok
en/introspect
```

Atelier 4. Sécurisation d'applications

4.1 Applications natives

4.1.1 loopback URI

Enregistrer un client :

- Client ID: ***cli***
- Access Type: ***public***
- Standard Flow Enabled: ***ON***
- Valid Redirect URIs: ***http://localhost/callback***

Récupérer les sources fournis, les visualiser et les comprendre :

```
npm install
npm start
```

S'authentifier sur la page de Keycloak, l'application doit afficher :

- Son port d'écoute
- Le code d'autorisation
- Le jeton obtenu

4.1.2 Device Code Flow

Modifier le client ***cli*** afin qu'il supporte le grant type ***Device***

Envoyer une requête POST sur le *device_authorization_endpoint*

```
curl -X POST \
    -d "client_id=cli" \
    "http://localhost:8080/realms/formation/protocol/openid-connect/auth/device"
```

Cela retourne une réponse comme suit :


```
{
"device_code": "MABcN2jmFFn8kvy-7w2idCnF4lZcTtmzkFbB_Nz4kHU",
"user_code": "WEWK-HKCH",
"verification_uri": "http://localhost:8080/realms/formation/device",
"verification_uri_complete": "http://localhost:8080/realms/formation/device?
user_code=WEWK-HKCH",
"expires_in": 600,
"interval": 5
}
```

Ouvrir le navigateur à ***verification_uri***. Enter le ***user_code***, s'authentifier et consentir.

Finalement, obtenir le jeton via :

```
curl -X POST \
    -d "grant_type=urn:ietf:params:oauth:grant-type:device_code" \
    -d "client_id=cli" \
    -d "device_code=<device_code>" \
    "http://localhost:8080/realms/formation/protocol/openid-connect/token"
```

4.2 Client Credentials Flow

4.2.1 Client credentials avec secret

Créer un nouveau client :

- Client ID: ***service***
- Client Protocol: ***openid-connect***
- Client authentication: ***ON***
- Standard Flow Enabled: ***OFF***
- Implicit Flow Enabled: ***OFF***
- Direct Access Grants Enabled: ***OFF***
- Service Accounts Enabled: ***ON***

Obtenir les jetons avec la commande curl suivante :

```
curl --data
"client_id=service&client_secret=$SECRET&grant_type=client_credentials"
http://localhost:8080/realms/formation/protocol/openid-connect/token
```

4.2.2 Client Credentials et X509

4.2.2.1 Création du client

Dans le realm ***formation***, créer un client ***svc-client***

- Type : ***OpenID Connect***
- Client authentication : ***true***
- Authentication Flow : ***Service Account roles***

Accéder à l'onglet credentials et modifier Client Authenticator à ***X509 certificate.***

Indiquer le subject DN : ***CN=alice,O=PLB,L=Paris,ST=IDF,C=FR***

4.2.2.2 Test de la configuration

Dans le répertoire ou se trouvent le certificat et la clé privé, exécuter la commande curl suivante :

```
curl -v -k \
  --cert user.crt \
  --key user.key \
  -X POST https://localhost:8443/realms/formation/protocol/openid-connect/token \
  -d grant_type=client_credentials \
  -d client_id=svc-client
```

Atelier 5. Intégration

5.1 Intégration javascript

Utiliser le client ***frontend*** et exporter la configuration du client dans un fichier ***keycloak.json***

Clients → frontend → Bouton Action → Download Client Adapter

Récupérer le code fourni, placer le fichier *keycloak.json* à la racine.

Compléter le code afin

- d'initialiser keycloak :
  - La page de login Keycloak s'affiche si l'utilisateur n'est pas authentifié
  - PKCE
- Utiliser le jeton d'accès dans l'appel au service

L'application backend est une application Node.js qui utilise l'adaptateur Keycloak *(keycloak-connect)*

- Exporter la configuration du client backend,
- Récupérer le code fourni, le regarder et démarrer le backend

Faire en sorte que le front-end puisse invoquer le backend

5.2 Intégration Spring, Appli Web traditionnelle

Créer un client ***springapp*** en autorisant le flow standard.

Préciser les URLs de redirection valides

Récupérer le projet SpringBoot qui inclut les starters suivants :

- webflux
- security
- oauth2-client

La classe principale :

- configure le login oAuth2
- Ajoute un endpoint qui renvoie l'utilisateur loggé

Déclarer le client ***springapp*** dans Keycloak avec :

- Client authentication
- Standard Flow
- Root URI, HomePage URI : http://localhost:8001
- Valide redirect URI : http://localhost:8001/login/oauth2/code/keycloak
- Logout URI et WebOrign : +

Reporter le client secret dans ***src/main/resources/application.yml***

Démararr l'application

Accéder au endpoint http://localhost:8001/oidc-principal et visualiser les *Authority* spring de l'utilisateur authentifié.

5.3 Intégration Quarkus

Arrêter le serveur KeyCloak utilisé jusqu'à maintenant, nous allons utiliser le DevService de quarkus

Récupérer le projet *delivery-service* fourni, regarder ses dépendances et le code source en particulier la dépendance sur ***quarkus-oidc***
Tester un démarrage en profil développement avec
```
./mvnw quarkus:dev
```

Vérifier du démarrage du container Keycloak via :
```
$ docker ps
```

Visualiser le code du contrôleur ***org.formation.web.LivraisonController*** et les contrôles d'accès indiqués par l'annotation ***@RolesAllowed***

Accéder à la DevUI :
http://localhost:8000/q/dev/

Puis au lien OpenIDConnect

Accéder à l'administration Keycloak via le lien proposé
Se connecter à l'administration avec ***admin/admin***
Visualiser la configuration de l'application ***quarkus-app*** et ses client scope

A partir de la DevUI, activer le bouton de login

Se logger avec ***bob/bob***
Observer le token d'accès

Accéder à une URL GET (vous pouvez utiliser le lien swagger)
Essayer un POST
Observer les requêtes envoyées avec F12 et visualiser le jeton envoyé dans la requêtes

Se logger avec ***alice/alice*** et essayer l'URL POST permettant de créer une livraison

Reprendre le projet ***notification-service***, visualiser ses dépendances vers *oidc* et la protection de ses endpoint via le rôle ***admin***

Décommenter dans ***deliveryservice*** la ligne 80 de
org.formation.service.impl.LivraisonServiceImpl.java
Cette ligne fait un appel rest vers notification-service

Se logger avec Alice sur ***delivery-service*** et essayer de créer une livraison, vous devez obtenir un 401

Ajouter les extensions suivantes au projet delivery-service

```
<dependency>
	<groupId>io.quarkus</groupId>
	<artifactId>quarkus-oidc-token-propagation-reactive</artifactId>
</dependency>
<dependency>
	<groupId>io.quarkus</groupId>
	<artifactId>quarkus-oidc-client-reactive-filter</artifactId>
</dependency>
```

Activer le filtre ***AccessTokenRequestReactiveFilter*** sur le Client REST :
Dans org.formation.service.NotificationService ajouter l'annotation
`@RegisterProvider(AccessTokenRequestReactiveFilter.class)`

Cette annotation propage le jeton.

En se loggant avec alice et en essayant de créer une livraison, on doit obtenir 201

Atelier 6 : Stratégies d’autorisation

6.1. GBAC

Choisir le client *backend*.

Ajouter un nouveau mapper avec les informations suivantes :

- Name: ***groups***
- Mapper Type: ***Group Membership***
- Token Claim Name: ***groups***

Créer ensuite un groupe ***« Human Resource »***puis un sous-groupe ***« Manager »***

Affecter l’utilisateur au groupe Manager

Retourner dans *backend → Client Scopes → Evaluate*

Sélectionner le user et générer le jeton d’accès

6.2. Autorisation services

Récupérer l’application *springboot* fournie

Dans cette application, il existe 2 chemins protégés par des autorisations spécifiques dans Keycloak :

- *****/api/{resource}***** : l'accès à cette ressource est basé sur l'évaluation des permissions associées à une ressource « *Default Resource* » dans Keycloak. tout utilisateur avec un rôle *user* est autorisé à accéder à cette ressource.
- *****/api/premium***** : l'accès à cette ressource est basé sur l'évaluation des permissions associées à une ressource « *Ressource Premium* » dans Keycloak. Seuls les utilisateurs avec un rôle *user-premium* sont autorisés à accéder à cette ressource.

Vous pouvez utiliser deux utilisateurs distincts pour accéder à cette application :

- ***alice/alice*** : utilisateur
- ***jdoe/jdoe*** : utilisateur, utilisateur-premium

La première étape consiste à créer un realm avec les utilisateurs, importer le fichier ***config/quickstart-realm-simple.json*** qui :

1. Crée le realm ***spring-boot-quickstart***
2. Crée les « realm roles » et les utilisateurs
3. Crée le client confidentiel ***app-authz-rest-springboot*** avec l’activation du service d’autorisation
4. Définit les 2 ressources « ***Default Resource*** » et « ***Premium resource*** »

Ensuite via la console d'admin, dans l'onglet autorisation du client *spring-boot-quickstart*

Créer 2 policies basées sur les rôles :

- **Only User Policy** : Rôle *utilisateur*
- **Only Premium Policy** : Rôle *utilisateur-premium*

Créer 2 permissions « resource based » associées au policies précédentes

- **Default Resource Permission**
- **Premium Resource Permission**

Démarrer l'application springboot :

```
mvn spring-boot:run
```

Obtenir un jeton d'accès, par exemple pour *alice* :

```
export access_token=$(\
    curl -X POST
http://localhost:8080/realms/spring-boot-quickstart/protocol/openid-connect/
token \
    -H 'Authorization: Basic YXBwLWF1dGh6LXJlc3Qtc3ByaW5nYm9vdDpzZWNyZXQ=' \
    -H 'content-type: application/x-www-form-urlencoded' \
    -d 'username=alice&password=alice&grant_type=password' | jq --raw-output
'.access_token' \
    )
```

Accéder aux ressources :

```
curl http://localhost:8180/api/resourcea \
  -H "Authorization: Bearer "$access_token
```

On doit obtenir la réponse ***« Access granted »*** ou un ***403*** si la ressource n'est pas permise

Échanger le jeton d'accès avec un jeton RPT

```
export rpt=$(curl -X POST \
 http://localhost:8080/realms/spring-boot-quickstart/protocol/openid-connect/
token \
 -H "Authorization: Bearer "$access_token \
 --data "grant_type=urn:ietf:params:oauth:grant-type:uma-ticket" \
 --data "audience=app-authz-rest-springboot" \
  --data "permission=Default Resource" | jq --raw-output '.access_token' \
 )
```

Visualiser le jeton JWT

Accéder aux ressources avec ce jeton

```
 curl http://localhost:8180/api/resourcea \
    -H "Authorization: Bearer ${rpt}"
```

Atelier 7 : Extension des standards

7.1 DPoP – Validation d'un appel API protégé

Objectif : Implémenter une protection DPoP sur une API et tester le rejet sans preuve

- Générer une clé (en JS)
- Obtenir un token depuis Keycloak avec DPoP activé
- Appeler une API sécurisée avec et sans en-tête DPoP
- Observer le rejet si le JWT est absent, ou que la clé ne correspond pas

Démarrer le docker compose du TP

- Importer le client ***dpop-client*** avec le json fourni ***dpop-client.json***

Vérifier la configuration dpop du client :
Advanced → Advanced settings

Visualiser le code su serveur d'API et le démarrer :

```
cd api-server
node api-server.js
```

Visualiser le code Javascript du client

Le code :

1. Génère une paire de clé
2. Calcule le thumbprint
3. Créé un DpoP pour obtenir un jeton
4. Obtiens un jeton d'accès en utilisant client_credentials et une entête DpOP
5. Recrée un DpOP pour accéder à l'api
6. Accède le service avec une requête DpOP
7. Accède le service avec un bearer simple

Exécuter et visualiser la console :

```
cd client
node dpop-client-credentials.js
```

7.2 Preuve de possession avec mTLS

Reprendre le client ***svc-client*** qui utilisait mTLS pour s'authentifier.

Lier le certificat au token :

Advanced → Advanced settings

Visualisez le code du serveur d'api fourni, il configure node.js en https et contient du code pour :

- Extrait le thumbprint du jeton
- Calcule le thumbprint attendu à partir du certificat
- Vérifie qu'il corresponde

Démarrer le serveur

```
npm install
node api-server.js
```

Obtenir un jeton auprès de Keycloak :

```
curl  -k   --cert user.crt   --key user.key  -X POST \
https://localhost:8443/realms/formation/protocol/openid-connect/to
ken   \
-d grant_type=client_credentials   -d client_id=svc-client
```

Positionner TOKEN en variable d'environnement :

```
export TOKEN=
```

Accéder au serveur d'API :

```
curl -v --cert user.crt --key user.key   --cacert ca.crt \
  -H "Authorization: Bearer $TOKEN" \
 https://localhost:9443/resource
```

Vérifier la sortie du serveur d'API

7.3 : Client policies

Dans ***Realm Settings – Client policies***

Définir un client profile ***pkce*** qui comporte l'exécuteur ***pkce-enforcer***

Définir une client policy ***pkce_4_all*** qui applique ce profil à tous les clients

Tester avec l'application frontend de l'atelier *3_oauth2* puis l'application frontend de l'atelier *5_Integration*

Atelier 8 : Exploitation

8.1 Administrateur dédié à un realm

Dans le realm ***formation***, créer un utilisateur ***admin-formation***, lui donner le rôle ***realm-admin*** sur l'application ***realm-management***, lui positionner le mot de passe admin.

Se connecter ensuite avec cet utilisateur à la console d'administration du realm formation :

http://localhost:8080/admin/formation/console

8.2 Intégration à un fournisseur d'identité OpenID Connect

Créer un realm ***third-party-provider***

Puis un client avec les paramètres suivants :

- ID : ***broker-app***
- Type : ***Confidential***
- rootURL : http://localhost:8080/realms/formation/broker/oidc/endpoint
- Valid redirect URI : *******

Créer également un utilisateur ***third-party-user***

Créer ensuite dans le realm *formation* une nouveau fournisseur d'identité en choisissant OpenID Connect

Indiquer dans l'URL de discovery celle du realm précédent :

http://localhost:8080/realms/third-party-provider/.well-known/openid-configuration

Indiquer le client ***broker-app*** et son secret

Accéder ensuite à la page de login des compte utilisateur :

http://localhost:8080/realms/formation/account

On doit proposer de se logger avec ***oidc***

8.3 Flow d'authentification

8.3.1 Usage de OTP

Télécharger sur votre mobile *FreeOTP (Redhat)* ou *Google Authenticator*, configurer l'utilisateur user afin qu'il utilise OTP

Tester à nouveau une séquence de login

8.3.2 Usage de WebAuthn

Configurer un flow permettant le login par user /mot de passe ET (WebAuthn. ou OTP)

Se déconnecter et essayer de l'utiliser pour se connecter

Créer un flux d'authentification ***Browser MFA*** comme suit :

- ***Cookie / Alternative***
- ***Kerberos***
- ***Identity Provider Redirector*** / ***Alternative***
- Un sous flux ***Formulaires*** / ***Alternative***:
  - ***Username Form*** / **Required**
  - Un sous-flux ***Authentification*** / ***Required***
    - ***Webauthn Authenticator*** / ***Alternative***
    - ***OTP Form / Alternative***

Se connecter sur la console avec l'utilisateur user avec Chrome et enregistrer une **Security Key**

Utiliser ce flow pour l'application ***account-console*** :
Client – Adavanced – Authentification flow overrides

Tester l'authentification avec WebAuthn

8.3.3 Usage du claim acr

Définir un mapping de acr_values comme suit :

```
urn:formation:pwd = 1
urn:formation:mfa =2
```

Un flow comme suit :

```
├─ auth-cookie (ALTERNATIVE)
├─ identity-provider-redirector (ALTERNATIVE)
└─ Forms Sub-flow (ALTERNATIVE)
    ├─ Username Password Form (REQUIRED)
    └─ MFA Conditional Sub-flow (CONDITIONAL LoA=2)
```

```
        ├─ Level of Authentication Condition (LoA = 2)
        ├─ OTP Form (ALTERNATIVE)
        └─ WebAuthn Authenticator (ALTERNATIVE)
```

Utiliser l’application cliente javascript fourni pour voir saisir une claim ***acr_values*** au moment de la requête OIDC

Atelier 9 : Développement serveur

9.1 Admin REST API

Récupérer un jeton admin :

- Soit en utilisant Password Credentials et le client admin-cli
- Soit en en se créant une application dédiée et le grant type Client Credentials

Effectuer des requêtes sur le realm ***formation*** :

- Liste de USER
- Création de USER
- Ajout de rôle à un USER
- Liste des Mappers pour un client donné

Documentation :

https://www.keycloak.org/docs-api/22.0.1/rest-api/index.html

Idem avec *kcadm*

9.2 Thème

Démarrer la stack fourni, elle monte le répertoire themes au bon endroit dans keycloak et démarre keycloak sans activer le cache

Créer un répertoire ***mytheme***

Y placer un fichier theme.properties avec :

```
parent=base
import=common/keycloak
```

Se connecter en admin, sur le realm master sélectionner le thème mytheme pour le login

Accéder à

http://localhost:8000/realms/master/account

Visualiser la page de login

Editer le fichier theme.properties et changer base par keycloak

Actualiser la page de login

Sous le répertoire mytheme/login créer un répertoire resources/css

Y créer un fichiers styles.css avec :

```
.login-pf body {
    background: DimGrey none;
}
```

Ajouter la ligne suivante dans theme.properties :

styles=css/styles.css

Actualiser la page puis remplacer la ligne par

styles=css/login.css css/styles.css

Éditer ensuite le style pour remplacer le logo keycloak par une de vos images

9.3 Event Listener SPI

Option 1 : Sans docker

Créer un projet Maven avec le ***pom.xml*** fourni

Implémenter l'interface ***EventListenerProvider*** en particulier les 2 méthodes :

- `public void onEvent(Event event)`
- `public void onEvent(AdminEvent event, boolean includeRepresentation)`

en affichant un message sur la console

Implémenter l'interface ***EventListenerProviderFactory*** en particulier les méthodes :

- `public EventListenerProvider create(KeycloakSession session)`
- `public String getId()`

Créer un fichier ***resources/META-INF/services/org.keycloak.events.EventListenerProviderFactory*** référençant votre classe factory

Construire le projet

```
mvn clean install
```

Copier le jar dans /provider

Effectuer un build

Redémarrer le serveur Keycloak

Dans la console d’administration, dans l’onglet ***Events Config*** ajouter votre listener.

Effectuer des opérations de login logout et surveiller la console

Option 2 : Avec docker

Observer le fichier docker-compose fourni, son montage de volume et sa commande de démarrage.

Démarrer la stack,

Se connecter sur la console d’administration et configurer l’EventListener ***sysout***

Realm Settings → Events → Events listener

Atelier 10 : Vers la production

10.1 Image optimisée

Récupérer et visualiser le fichier *Dockerfile* fourni, l'adapter si besoin

Construire une image Docker :

```
$ docker build -f keycloak-formation .
```

Visualiser ensuite le fichier *docker-compose* fourni, l'adapter si besoin

Démarrer la stack

Vérifier le démarrage des 3 services

Accéder à *pgAdmin* (localhost:81)

Enregistrer le serveur Postgres :

- host : ***keycloak-postgresql***
- user/password : ***postgres/postgres***

Visualiser les tables de la base ***keycloak***

Accéder à Keycloak avec l'utilisateur admin

Exporter le realm formation de l'environnement de dev :

Realm Settings → Action → Partial Export

Puis essayer recréer le realm à partir de l'export dans le keycloak de production