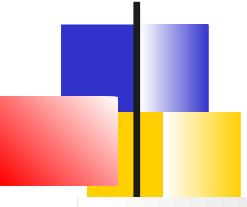


Gestion centralisée de la sécurité avec Keycloak

David THIBAU - 2025

david.thibau@gmail.com

Keycloak - Identity and Access Management for Modern Applications
Stian Thorgersen | Pedro Igor Silva



Agenda

Introduction

- Fonctionnalités, distribution et installation, UIs Admin et Utilisateur, Sécurisation 1ère application

Rappels sur les standards

- OAuth2.0, OpenID Connect, JWT, types de jetons

Authentification avec OpenIdConnect

- Discovery, Authentification (Browser, X.509), Flow CIBA, Jeton d'identification et userinfo endpoint, Logout

Autorisation avec oAuth2

- Authorization Code Flow, Limitations des accès, Validation du jeton

Sécurisation des différents types d'application

- Application web, Application native ou mobile, Services back-end

Intégration Keycloak

- Librairies et adaptateurs, SpringBoot, Quarkus, Reverse Proxy

Stratégies d'autorisation

- RBAC, GBAC, OAuth2 scopes, Authorization service

Maximiser la sécurité des standards

- Modèles des attaques, recommandation RFC 9700, modules DpoP et FIPS, FAPI et Tests de conformité

Administration Keycloak

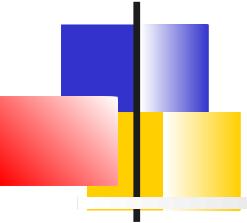
- Gestion des utilisateurs, Authentification des utilisateurs, Gestion des sessions et jetons

Développement Serveur

- Admin API, Thème, SPI

Recommandations pour la production

- Configuration de production, Sécurisation Keycloak, Comptes utilisateurs et Applications

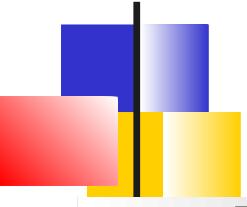


Introduction

Fonctionnalités et installation

Interfaces Admin et utilisateur

Sécuriser une 1ère application



Keycloak

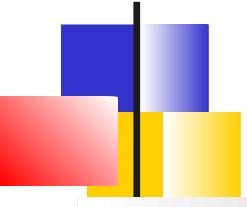
Produit OpenSource pour la gestion d'identité et des accès (IAM) dédié aux applications modernes (SPA, Mobile, API Web) :

=> Les applications n'ont plus besoin d'implémenter l'authentification ni le stockage de mot de passe

=> On obtient du SSO à l'intérieur de l'entreprise

Projet démarré en 2014, largement déployé en entreprise depuis

- Hautement personnalisable, Support pour l'authentification forte, application de stratégies pour les comptes utilisateur
- S'appuie sur les standards oAuth2.0, OpenIDConnect et SAML
- S'intègre avec des fournisseurs d'identité tierces : LDAP, OpenID Provider, ...

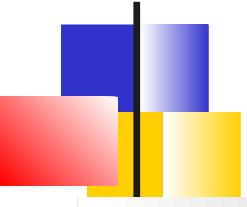


Base utilisateurs

Keycloak est livré avec sa propre base de données d'utilisateurs¹

Il est possible d'intégrer une infrastructure d'identité existante :

- Bases d'utilisateurs existantes à partir de réseaux sociaux
- Fournisseurs d'identité d'entreprise (Autre Keycloak par exemple)
- Annuaires d'utilisateurs existants Serveurs Active Directory et LDAP.



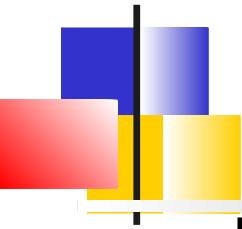
Stack technologique

Avant Fin 2020: Application web déployée sur le serveur Wildfly

Après Fin 2020 : Construit au-dessus du framework Quarkus¹ :

- Temps de démarrage réduit
- Faible empreinte mémoire
- Approche conteneur
- Meilleure expérience de développeur
- Meilleure convivialité

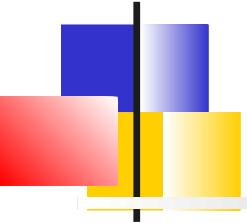
1. Une pile native Kubernetes et Cloud avec les meilleures bibliothèques et standards Java



Installation

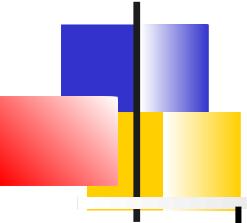
Différentes options pour l'installation :

- Installation locale via une JVM
- Démarrage de conteneur *Docker* ou *Podman*
- Déploiement sur *Kubernetes* ou *OpenShift*
avec en option l'utilisation de *Keycloak Kubernetes Operator* qui facilite l'installation, la configuration et l'exploitation



Introduction

Fonctionnalités et installation
Interfaces Admin et utilisateur
Sécuriser une 1ère application



Console d'administration

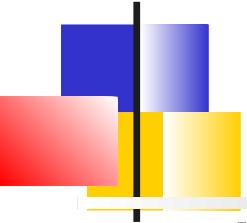
La console d'administration permet de gérer des **realms** :

- Ensemble d'utilisateurs
 - Rôles, groupes, sessions
- Ensemble d'applications clientes
 - Mécanismes d'obtention de jetons : Grant flows
 - Scopes : Accès aux données utilisateur

Un realm est complètement isolé des autres realms

- Par exemple, un realm pour les applications internes et les employés, et un autre pour les applications externes

Le realm *master* détermine la sécurité du serveur KeyCloak



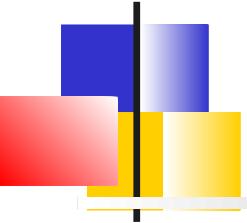
Création de realm

Le formulaire de création de realm ne demande qu'un nom

Le nom est utilisé dans les URLs (Pas d'espace ni de caractères spéciaux)

- Une nom d'affichage peut également être précisé

Un *realm* peut s'exporter/s'importer via le format JSON



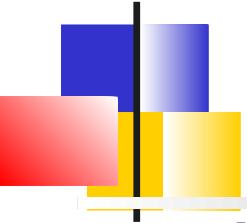
Création d'utilisateur

Après la création de *realm*, on crée des utilisateurs :

- *username* <=> login
- email, nom et prénom
- Actions requises au premier login : Vérification de son profil et de son email par exemple

On peut ensuite compléter les attributs keycloak par des attributs personnalisés

Avant que l'utilisateur puisse se logger, il faut définir un mot de passe (temporaire ou non)

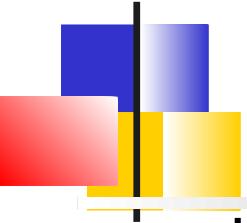


Création de groupe

Un **groupe d'utilisateur** permet de mutualiser des attributs ou rôles aux utilisateurs appartenant au groupe.

Un groupe est donc :

- Un nom
- Une liste d'attributs
- Éventuellement une liste de rôles

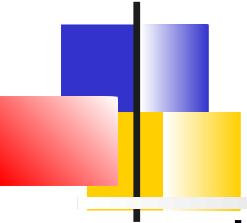


Création de client

Un client représente une application intégrée à Keycloak.

Lors de la création d'un client, on indique :

- Son nom
- Si il doit être authentifié
- Le flow d'interaction avec Keycloak : *Authorization Code, Client Credentials, Device Flow, CIBA*
- Son usage ou pas du service d'autorisation de keycloak
- Les scopes possibles (accès aux données utilisateurs)
- Les rôles attribuables spécifique de ce client



Création de rôle

Un **rôle** est généralement utilisé par les applications clientes pour déterminer les droits d'accès.

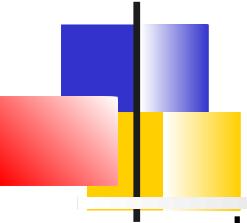
Au niveau de *KeyCloak*, un rôle est :

- Un nom
- Une description

Il existe un espace de noms global pour les rôles (realm roles) et un espace de nom dédié à un client.

Un rôle peut être associé à des utilisateurs ou des groupes

Un rôle peut être composite et donc contenir d'autres rôles.



Interface utilisateur

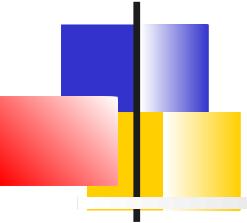
Les utilisateurs accèdent à Keycloak pour gérer leur propre compte.

L'URL d'accès est :

<http://<serveur>/realms/<realm-name>/account/>

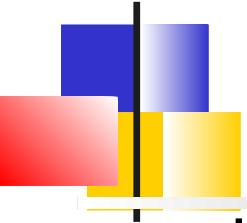
Par exemple :

- Mettre à jour son profil d'utilisateur
- Mettre à jour son mot de passe
- Activer l'authentification à deux facteurs
- Lister ses applications, les applications auprès desquelles ils se sont authentifiés
- Lister les sessions ouvertes, y compris la déconnexion à distance d'autres sessions



Introduction

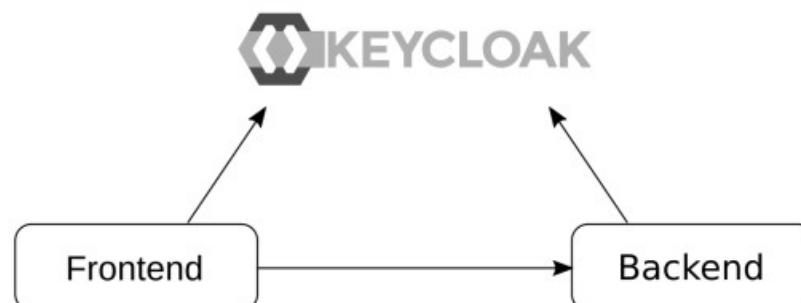
Fonctionnalités et installation
Interfaces Admin et utilisateur
Sécuriser une 1ère application



Exemple : SPA + API Rest

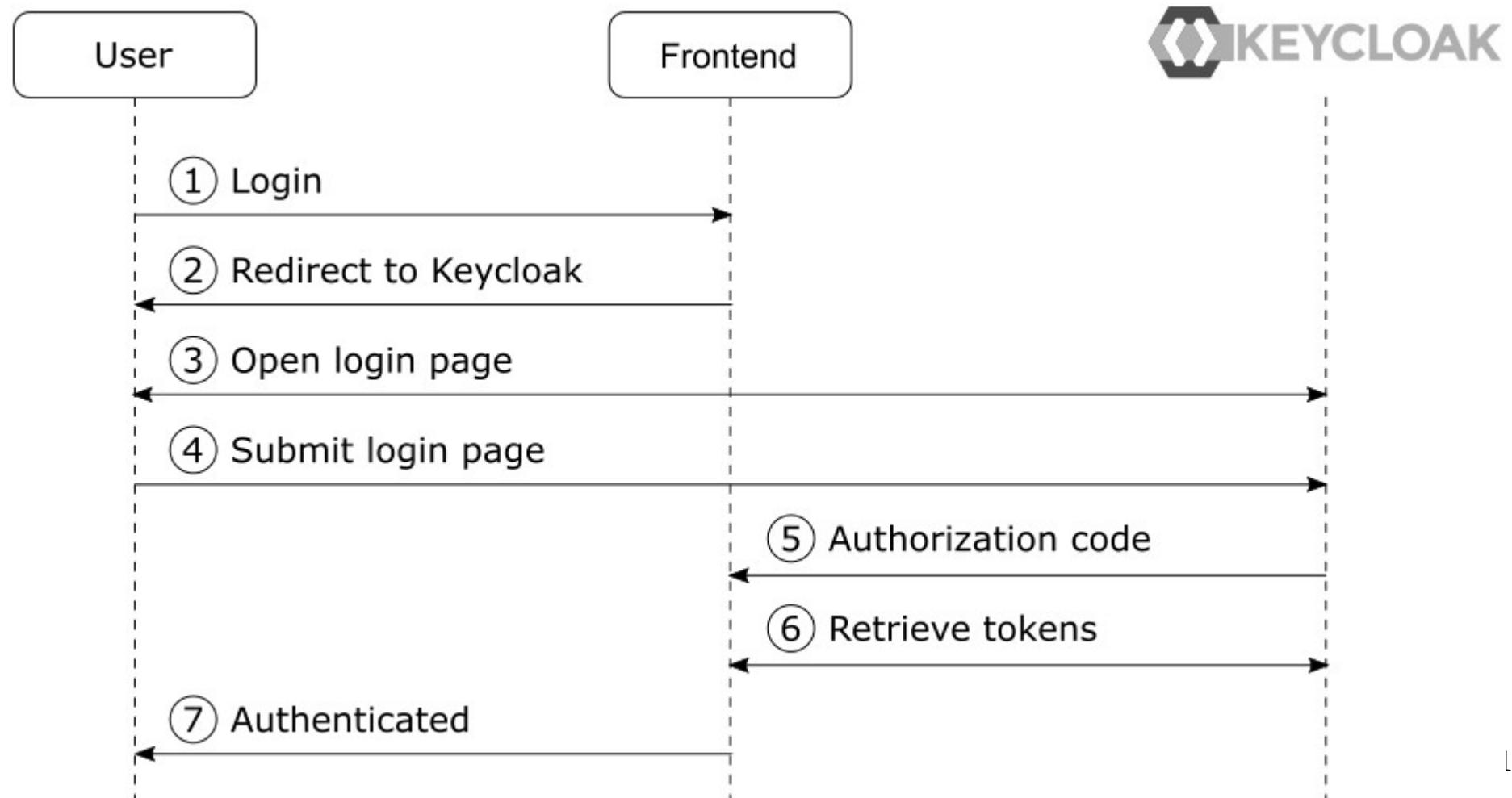
L'application exemple est composée de 2 parties :

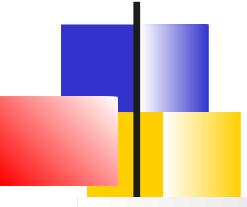
- une **front-end** type SPA
 - Authentifiant l'utilisateur via Keycloak afin d'obtenir des jetons le représentant : ID, accès et rafraîchissement
 - Utilisant le jeton d'accès pour invoquer une API Rest
- un **backend** dont les ressources sont protégées par oAuth2



Authentification FRONT-END

OpenID / Authorization code flow





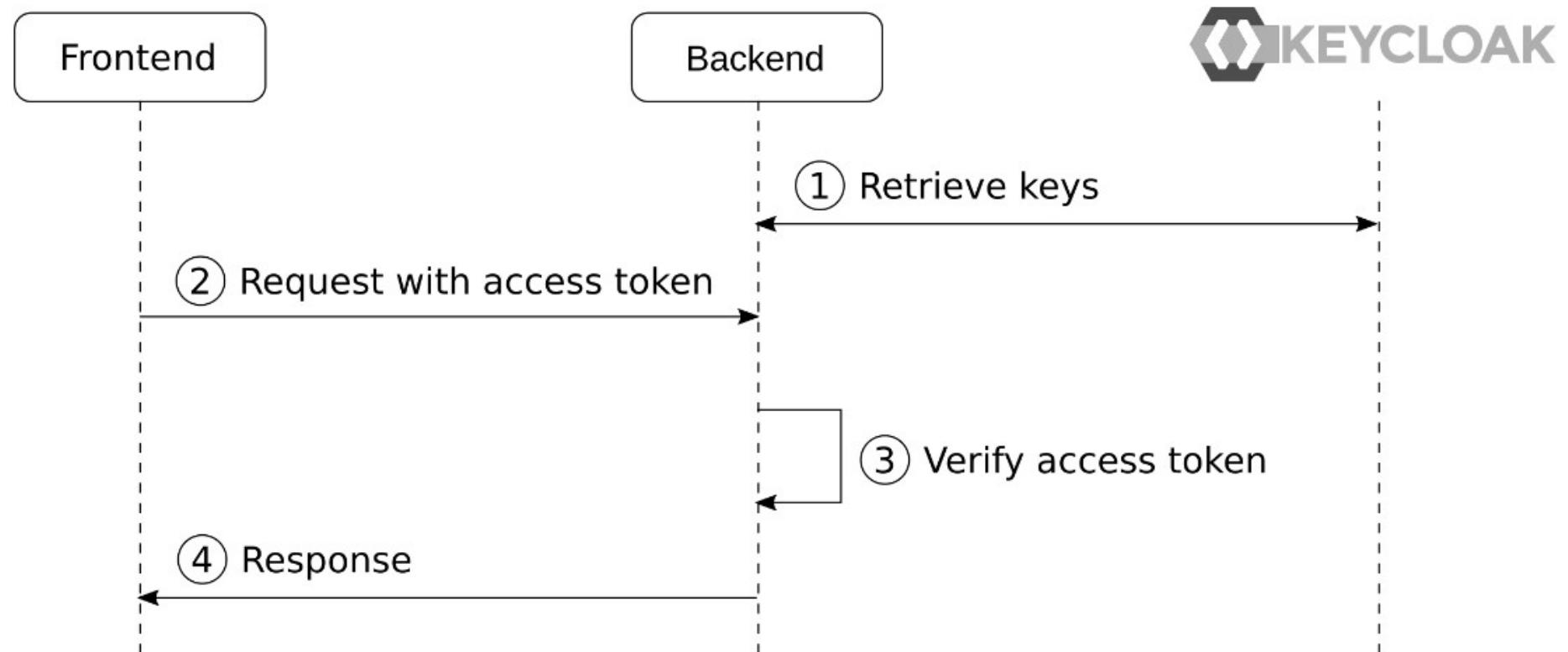
Claims Jetons

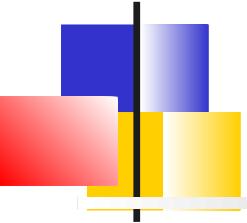
Les jetons obtenus contiennent des **claims** dépendant du **scope** demandé. Les jetons sont hautement personnalisables :

- ***ID_TOKEN*** : Établit l'identité de l'utilisateur
 - ***exp*** : Date d'expiration
 - ***iss*** : Issuer
 - ***sub*** : Identifiant unique de l'utilisateur
 - ***name, preferred_name, ...***
- ***ACCESS_TOKEN*** : Détermine les droits d'accès
 - ***allowed-origins*** : Pour le CORS
 - ***realm-access*** : Intersection entre les rôles de l'utilisateur et les rôles accessibles par le client
 - ***resource_access*** : Les rôles du client
 - ***scope*** : les scopes initiaux demandés

Vérification des accès BACKEND

JSON Web Signature (JWS)





Rappels sur les standards

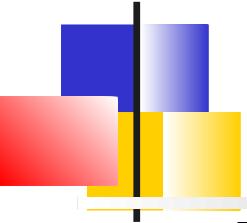
OAuth2

OpenID Connect

JWT

Transmission et protection des jetons

FAPI

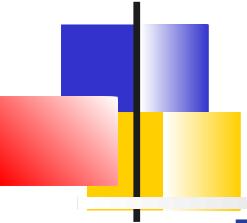


Apports de oAuth2

Protocole d'autorisation défini pour

- Le partage de données utilisateur entre applications
- Le contrôle les données partagées.

Au départ, orienté applications web grand public, il peut être appliqué aux applications d'entreprise pour contrôler les accès



Rôles définis dans oAuth2

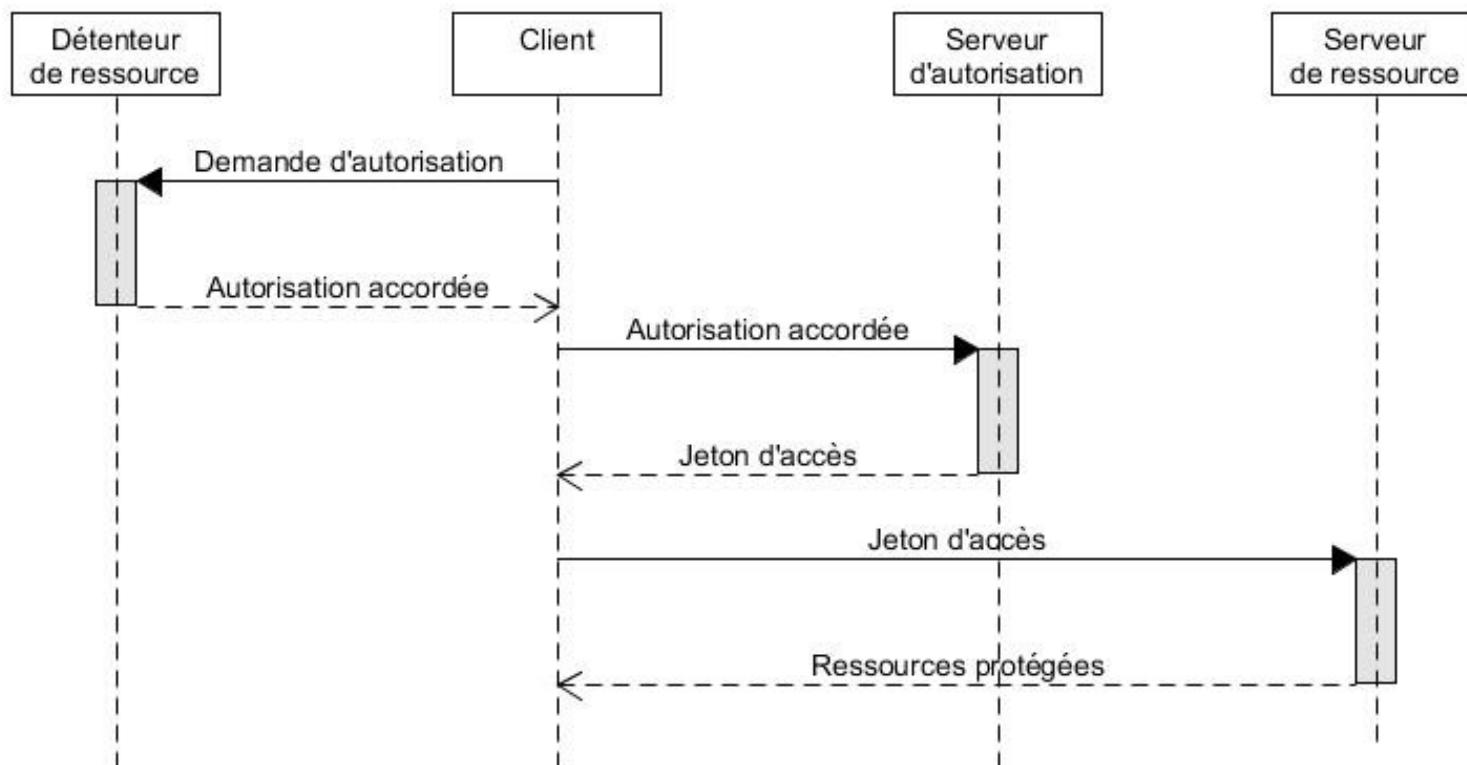
Propriétaire de la ressource : il s'agit généralement de l'utilisateur final qui possède les ressources auxquelles l'application veut accéder.

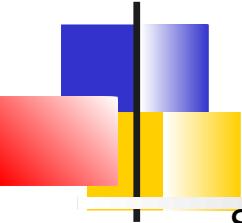
Serveur de ressources : C'est le service hébergeant les ressources protégées.

Client : Il s'agit de l'application qui souhaite accéder à la ressource

Serveur d'autorisation : C'est le serveur qui délivre l'accès au client. (Keycloak)

Étapes du protocole

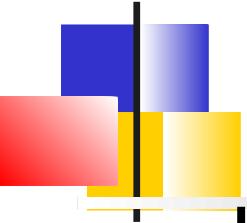




Obtention de jetons les différents flux ou Grant type

Selon le type d'application et le contexte d'usage, différents flows OAuth 2.0 permettent d'obtenir un jeton d'accès. Certains sont recommandés, d'autres à éviter.:

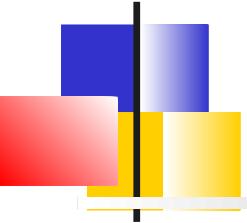
- **Authorization Code** : Le plus utilisé, l'utilisateur consent et est authentifié chez le fournisseur de jeton. Un code est alors renvoyé à l'application client qui l'utilise pour obtenir le jeton
Adapté aux applications web, mobiles, SPA (avec PKCE).
- **Client Credentials** : L'application cliente détient un secret qui lui suffit pour obtenir le jeton. L'application accède à la ressource en son propre nom (l'application est le détenteur de la ressource).
Adapté aux backend consommant des APIs
- **Device Flow** : L'application s'exécute sur un appareil sans browser et ne peut pas détenir un secret
Recommandé pour les cas "sans navigateur" et clients publics
- Le **Password flow** implique que l'application client obtienne les mots de passe de l'utilisateur
Déprécié , à éviter.
- **L'Implicit flow** défini dans la spéc est à éviter car il expose le jeton. Identique à Authorization Code Flow mais sans code d'autorisation, retour direct du jeton.
Déprécié, à éviter.



Types de clients

Les flux sont donc choisis en fonction du type de client.

- Les clients **confidentiels** sont des applications pouvant stocker des crédentiels en toute sécurité.
EX : Application back-end
- Les clients **publics**, en revanche, sont des applications exécutées côté client qui ne sont pas en mesure de stocker des crédentiels.
En général, ils s'exécutent dans le navigateur

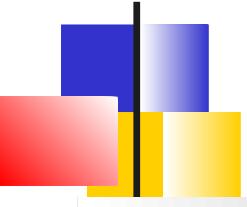


Protections pour les clients publics

Les clients publics s'exécutant dans un navigateur utilisent ***l'Authorization Code Flow***

2 mesures de protection supplémentaires sont en place :

- ***Valid redirect URI*** : Le serveur d'autorisation n'enverra le code permettant de récupérer un jeton seulement sur une URI prédéfinie.
- ***Proof Key for Code Exchange (PKCE, RFC 7636)*** : Extension d'OAuth 2.0 qui empêche quiconque ayant intercepté un code d'autorisation de l'échanger contre un jeton d'accès.



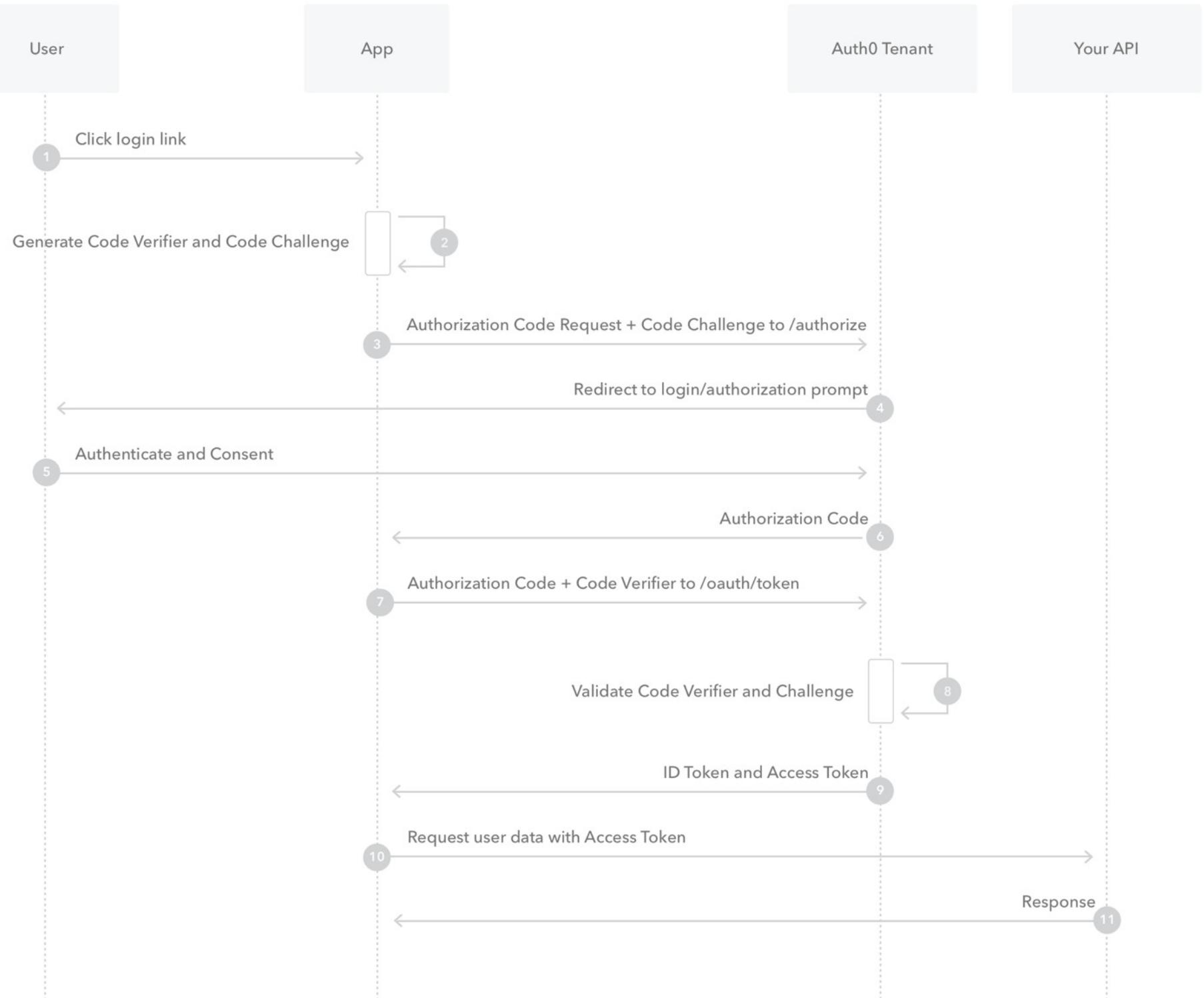
Flux avec PKCE

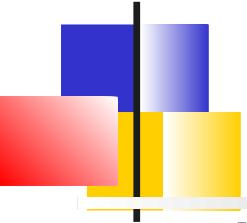
L'application cliente publique crée une chaîne aléatoire unique : le ***code_verifier***.

Le hash du *code_verifier* est ajouté à la requête de code d'autorisation sous le paramètre ***code_challenge***.

Une fois l'utilisateur authentifié et le code renvoyé, l'application demande les jetons avec le code d'autorisation et le *code_verifier*.

Si les codes correspondent, l'authentification est terminée et un *access_token* est renvoyé.

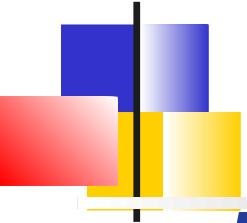




Validation des jeton

Lors de la réception du jeton, le serveur de ressource doit valider l'authenticité du jeton et extraire ses informations différentes techniques sont possibles

- Jetons opaque et Appel REST vers le serveur d'autorisation
- Jetons JWT et validation via paire de clés privé/publique

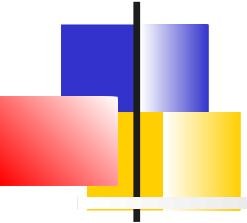


Spécifications additionnelles

Bearer tokens (RFC 6750) : Les jetons Bearer sont de loin le type de jetons d'accès le plus couramment utilisé, et ils sont généralement envoyés aux serveurs de ressources via l'en-tête *HTTP Authorization*.

Token Introspection (RFC 7662) : Dans OAuth 2.0, le contenu des jetons d'accès est potentiellement opaque pour les applications. Le endpoint d'introspection permet au client d'obtenir des informations sur le jeton d'accès sans comprendre son format.

Token Revocation (RFC 7009) : OAuth 2.0 prend en compte la manière dont les jetons d'accès sont délivrés aux applications, mais pas la manière dont ils sont révoqués. Ceci est couvert par le endpoint de révocation de jeton.



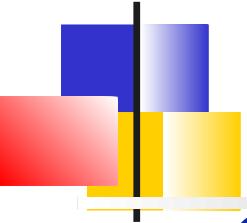
Rappels sur les standards

OAuth2
OpenID Connect

JWT

Transmission et protection des jetons

FAPI

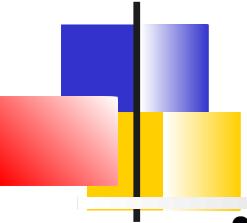


Introduction

OpenID Connect s'appuie sur OAuth 2.0 pour se concentrer sur l'authentification

Il apporte :

- Social login, (se logger avec son compte Google)
- SSO dans le cadre d'une entreprise
- Les applications clientes n'ont pas accès aux mots de passe des utilisateurs
- Il permet également l'utilisation de mécanismes d'authentification forte comme le *OTP (One Time Password)* ou *WebAuthn*

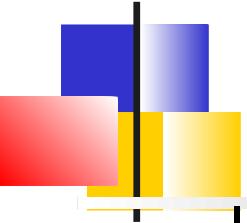


Rôles de OpenID

- **Utilisateur final** : Équivalent au détenteur de ressource dans OAuth 2.0. L'utilisateur qui s'authentifie.
- **Relying Party** (RP) : L'application qui souhaite authentifier l'utilisateur final équivalent à l'application cliente
- **Fournisseur OpenID** (OP) : Le fournisseur d'identité qui authentifie l'utilisateur. (Keycloak).

Dans un flux de protocole OpenID Connect, l'application demande l'identité de l'utilisateur final au fournisseur OpenID.

Comme il s'appuie sur OAuth 2.0, en même temps que l'identité de l'utilisateur est demandée, il peut également obtenir un jeton d'accès

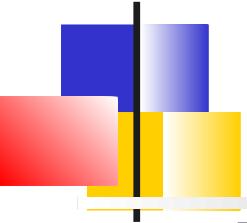


Flux d'OpenID

Il existe 3 flux dans OpenID Connect :

- **Flux avec code d'autorisation** : Comme OAuth 2.0, après identification sur le serveur d'autorisation, un code d'autorisation est envoyé à l'application. Il peut être échangé contre un jeton d'identification, un jeton d'accès et un jeton d'actualisation.
- **Flux hybride** : Dans le flux hybride, le jeton d'identification est renvoyé à partir de la demande initiale avec un code d'autorisation. Le code d'autorisation permet ensuite d'obtenir le jeton d'accès Beaucoup plus rare
- **Flux implicite** : Pas de code d'autorisation. déconseillé

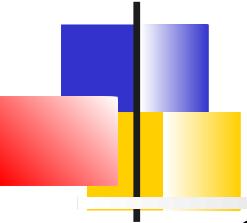
Ces 3 flux nécessitent un navigateur et un utilisateur qui interagit. Pas adaptés aux appareils sans navigateur, à un opérateur initiant une connexion pour l'utilisateur, App kiosk / TV, etc.



Flows sans navigateur

En absence de navigateur, d'autres flow oAuth2 sont possibles :

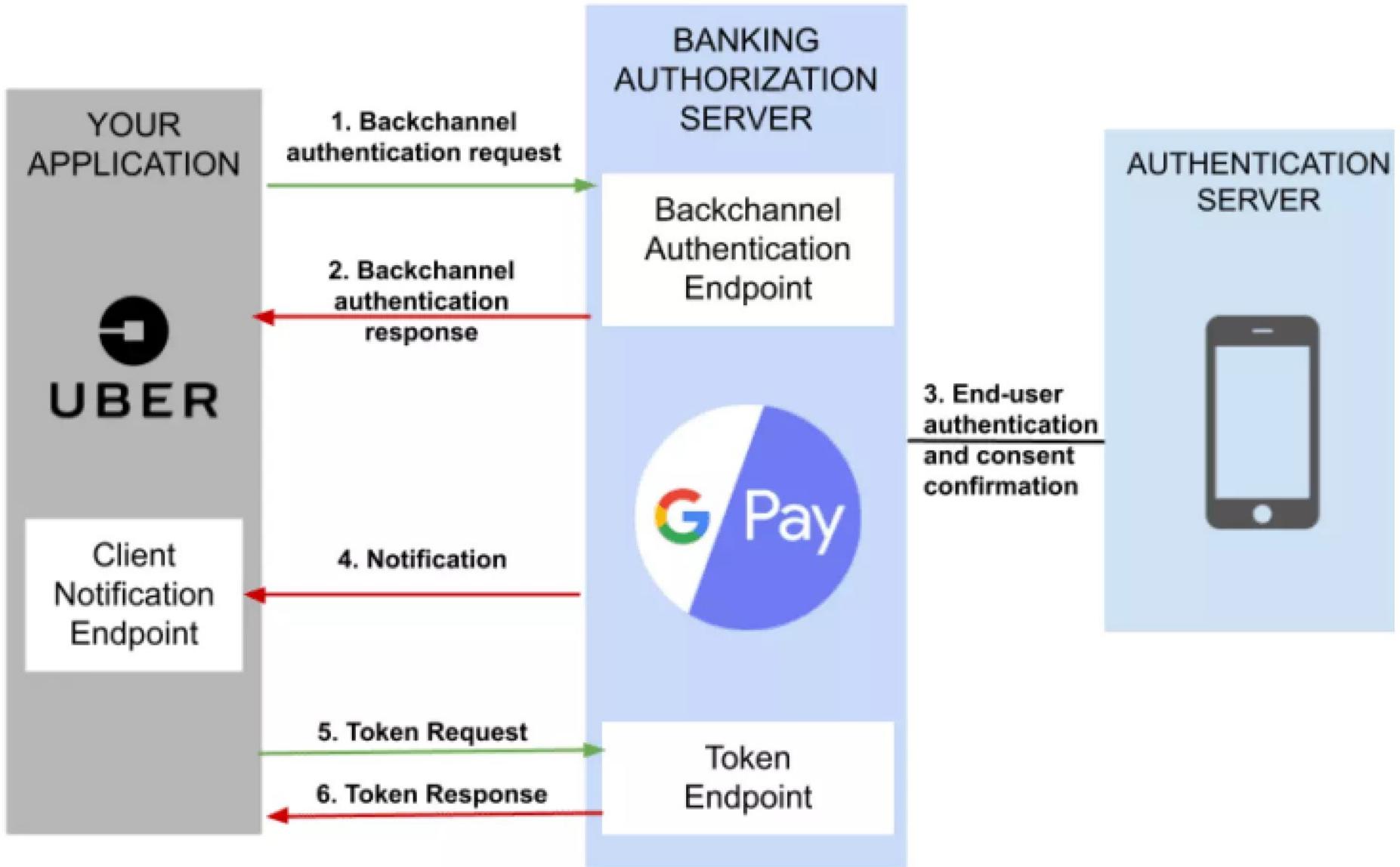
- Client-credentials : L'utilisateur est le client
- Device Authorization Flow : Navigateur externe à l'application
- CIBA Flow : Authentification déclenché par un backend

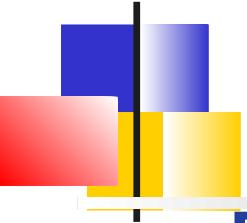


CIBA Flow

CIBA Flow ne repose pas sur un navigateur.

1. Le client (app, call center, TV, etc.) déclenche une demande d'authentification via un **appel backend** à l'Authorization Server
2. Le serveur contacte l'utilisateur via un **autre canal** (ex : push notification, app mobile, SMS).
3. L'utilisateur approuve ou rejette la demande d'authentification.
4. Le serveur notifie le client (via polling, webhook ou ping) une fois l'authentification terminée.
5. Le client récupère les jetons d'accès / d'identité.





Spécifications additionnelles OpenID

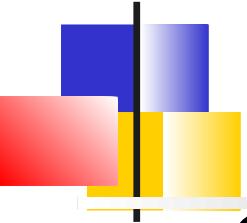
Discovery : Permet aux clients de découvrir dynamiquement des informations sur le fournisseur *OpenID*

Enregistrement dynamique : Permet aux clients de s'enregistrer de manière dynamique auprès du fournisseur OpenID

Gestion de session : Définit comment surveiller la session d'authentification de l'utilisateur final avec le fournisseur OpenID et comment le client peut initier une déconnexion

Front-Channel Logout : Définit un mécanisme de déconnexion simultanée de plusieurs applications à l'aide d'iframes intégrés

Back-Channel Logout : Définit un mécanisme de déconnexion simultanée utilisant un canal côté backend



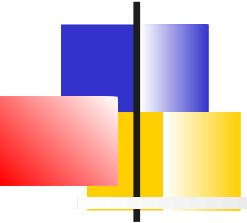
JWT et UserInfo Endpoint

OpenID Connect spécifie clairement le format **JWT** comme format du jeton d'identification

- Les valeurs dans le jeton (appelées claims) peuvent être lus directement par le client

OpenID définit également un **userinfo endpoint** qui peut être accédé avec un jeton d'accès.

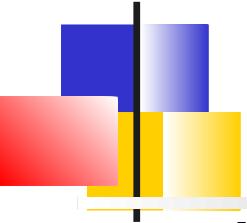
- Le endpoint fournit les mêmes informations que le jeton d'identification



Rappels sur les standards

OAuth2
OpenID Connect
JWT

Transmission et protection des jetons
FAPI

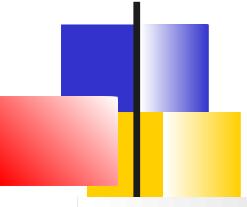


Introduction

Keycloak utilise **JWT** comme format par défaut pour les jetons d'accès également.

Les avantages de ce choix :

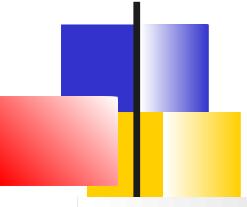
- Format JSON facilement parsable par les librairies
- Les serveurs de ressources peuvent lire les valeurs du jetons sans requête vers le serveur d'origine



JOSE

JWT est issu d'une famille de spécifications connue sous le nom de **JOSE**

- **JSON Web Token (JWT, RFC 7519)** : Le jeton se compose de 2 documents JSON encodés en base64 et séparés par un point : un en-tête et un ensemble de revendications (*claims*)
- **JSON Web Signature (JWS, RFC 7515)** : Ajoute une signature numérique de l'en-tête et des revendications
- **JSON Web Encryption (JWE, RFC 7516)** : Chiffre les revendications
- **JSON Web Algorithms (JWA, RFC 7518)** : Définit les algorithmes cryptographiques qui doivent être utilisés pour JWS et JWE
- **JSON Web Key (JWK, RFC 7517)** : Définit un format pour représenter les clés cryptographiques au format JSON



Récupération des clés JWKS

Le point de découverte défini par OpenID Connect permet de récupérer **l'ensemble de clés Web JSON (JWKS)** ainsi que les mécanismes de signature et de chiffrement de la spécification qui sont pris en charge.

Lorsqu'un serveur de ressources reçoit un jeton d'accès, il peut le vérifier en :

- Récupérant l'URL JWKS à partir du endpoint de discovery OpenID Connect.
- Téléchargeant des clés publiques de signature du fournisseur OpenID. Elles sont généralement mis en cache/stockées sur le serveur de ressources.
- Vérifiant la signature du jeton à l'aide des clés publiques.

jwt.io

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYOUT: DATA

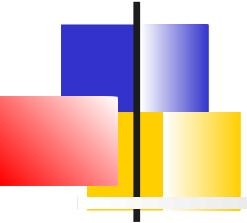
```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)  secret base64 encoded
```

✓ Signature Verified

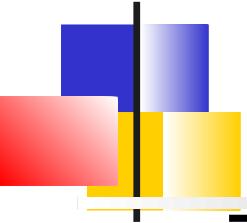
SHARE JWT



Rappels sur les standards

*OAuth2
OpenID Connect
JWT*

Transmission et protection des jetons
FAPI

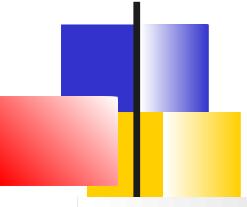


Introduction

Tous les jetons ne se valent pas en termes de sécurité.

Certains sont "présentables" (Bearer), d'autres nécessitent une preuve de possession (DPoP, mTLS), ou une validation côté serveur (Phantom, Split).

Keycloak prend en charge plusieurs de ces approches selon les cas d'usage.



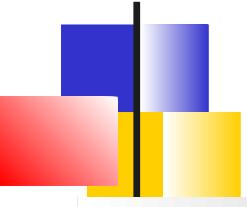
Bearer token

Jeton auto-suffisant, envoyé en clair dans l'en-tête HTTP :

Authorization: Bearer <token>

Le serveur de ressource accepte le jeton sans preuve supplémentaire.

- Supporté nativement par Keycloak.
- Risque : toute entité possédant le jeton peut l'utiliser (token replay).

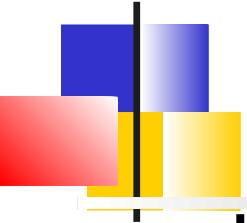


DPoP

Le client prouve qu'il détient une clé privée associée au jeton.

Envoie un header DPoP contenant une preuve cryptographique signée.

- Empêche l'usage d'un jeton volé par un tiers.
- Support partiel dans Keycloak : Features en preview

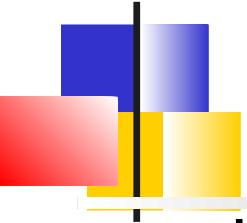


mTLS

Utilise une **authentification TLS mutuelle** (le client présente un certificat).

Le jeton est lié à l'empreinte du certificat.

- Très robuste, adapté aux cas réglementés (FAPI, banque).
- Supporté par Keycloak, mais nécessite :
 - Activation dans le client
 - Infrastructure compatible mTLS (certificats, terminaux)



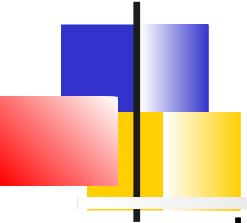
Split Token

Le jeton est divisé en deux parties :

- Une partie publique envoyée au client
- Une partie secrète stockée côté serveur

La vérification se fait en croisant les deux parties

! Non supporté nativement par Keycloak
→ Implémentable via des reverse proxies ou gateways.



Phantom Token

Le jeton envoyé au client est un alias opaque.

Le serveur de ressources interroge le serveur d'autorisation pour obtenir les informations.

- Avantage : le client ne voit jamais le vrai contenu du jeton.
- Non supporté dans Keycloak, mais :
peut être mis en place avec Token Introspection (RFC 7662) côté API gateway (ex : Apigee, Kong)

[Client] → Authorization: Bearer <phantom_token>

|

[API Gateway (Apigee)]

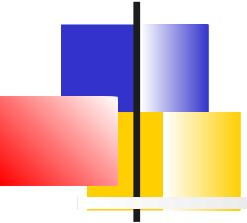
|

→ [Introspection avec Keycloak]

← { "active": true, "sub": ..., "scope": ... }

|

[Backend/API]

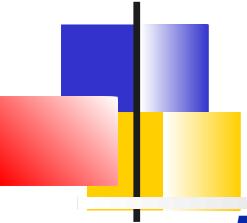


Rappels sur les standards

*OAuth2
OpenID Connect
JWT*

Transmission et protection des jetons

FAPI



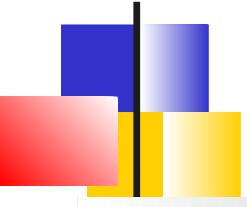
Financial Grade API

Financial-grade API (FAPI) est une spécification technique qui durcit OAuth2 et OpenID

- Publiée par l'OpenID Foundation, avec une adoption croissante dans les environnements réglementés (notamment PSD2¹ en Europe).

Objectifs principaux :

- Protéger contre les attaques de type interception de jeton, redirection malveillante, injection de code, etc.
- Renforcer l'authentification et la signature des requêtes.
- Garantir l'intégrité des messages et l'identité du client (preuve de possession).



Exigence FAPI

Flow OpenID : **Authorization Code + PKCE** obligatoire

Authentification client : **mTLS** ou **Private Key JWT¹**

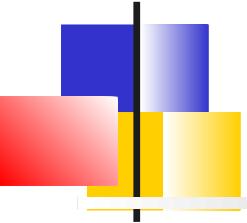
Jetons : Doivent être **signés et chiffrés**

ID Token : Doit inclure des **claims supplémentaires** (`acr`, `auth_time`, etc.)

Délai de validité : Jetons courts, rotation imposée

Preuve de possession : Recommandation : **DPoP ou mTLS**

1. Dans ce cas, client génère un JWT signé avec sa clé privée et l'envoie via le paramètre *client_assertion*



Keycloak et OpenID

Discovery endpoint

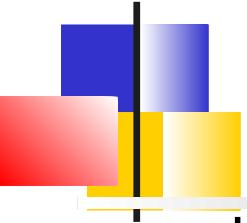
Authentification

Authentification via mTLS

CIBA Flow

Personnalisation du jeton

Logout



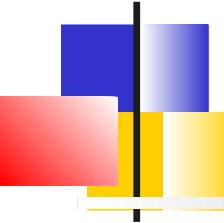
Discovery Endpoint

Le point de découverte de Keycloak accessible à :

<http://<server>/realms/<realm-name>/.well-known/openid-configuration>

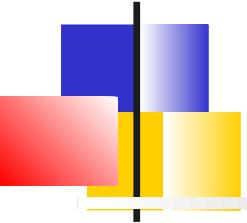
Permet à un client de découvrir tous les informations de configurations intéressantes

- Endpoints, URIs accessibles
- Types de grants supportés
- Types de réponses supportés (jeton, code, code+jeton...)
- Mécanismes de logout supportés
- Algorithmes de signatures et de chiffrement supportés
- Scopes supportés



Endpoints

```
"issuer":  
    "http://<server>/realms/<realm-name>,  
  
"authorization_endpoint":  
    "http://<server>/realms/<realm-name>/protocol/openid-connect/auth",  
  
"token_endpoint":  
    "http://<server>/realms/<realm-name>/protocol/openid-connect/token",  
  
"introspection_endpoint":  
    "http://<server>/realms/<realm-name>/protocol/openid-connect/token/introspect",  
  
"userinfo_endpoint":  
    "http://<server>/realms/<realm-name>/protocol/openid-connect/userinfo",  
  
"end_session_endpoint":  
    "http://<server>/realms/<realm-name>/protocol/openid-connect/logout",  
  
"jwks_uri":  
    "http://<server>/realms/<realm-name>/protocol/openid-connect/certs",
```



Authentification avec OpenID

Discovery endpoint

Authentification

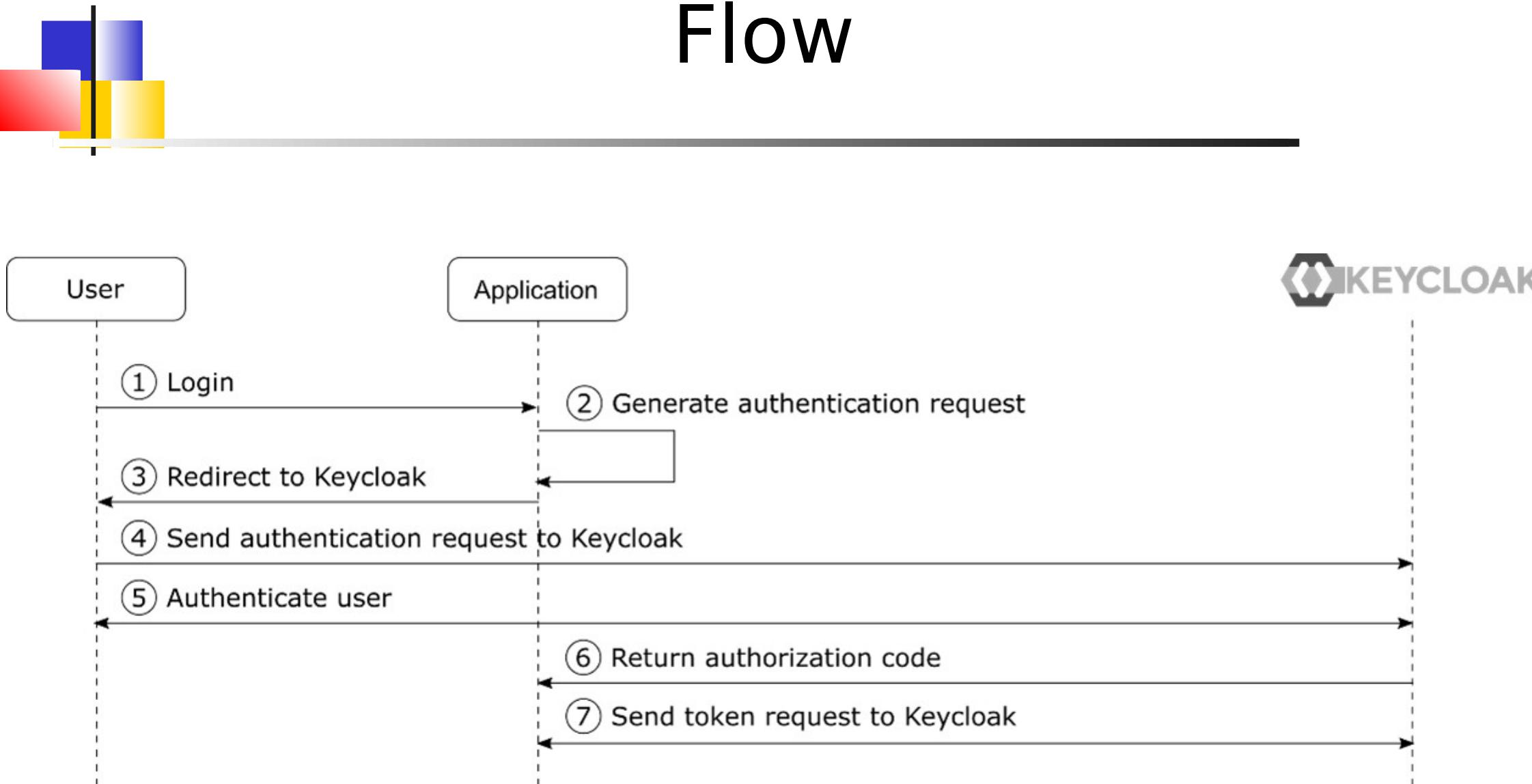
Authentification via mTLS

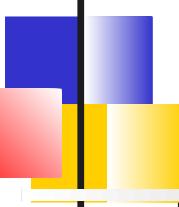
CIBA Flow

Personnalisation du jeton

Logout

Flow





Requête d'autorisation

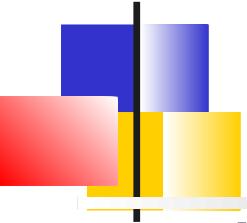
Les paramètres de la requête d'authentification doivent contenir :

- ***client_id*** : L'id du client
- ***redirect_uri*** : L'URI permettant à l'application de récupérer le code
- ***scope*** : Valeur par défaut ***openid***. Plusieurs scopes peuvent être précisés
- ***response_type*** : ***code*** (pour le code d'autorisation)

Elle peut contenir également :

- ***prompt*** (optionnel) : Conditionne la page de login (en fonction si l'utilisateur est déjà authentifié ou pas)
- ***max_age*** : Nombre de secondes maximum afin qu'une authentification précédente puisse être ré-utilisée
- ***login_hint*** : Possibilité de passer le *username*, si l'application le connaît

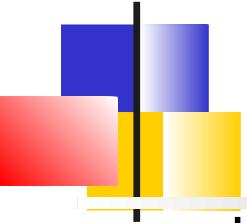
La réponse est une code d'autorisation valable pendant 1 mn par défaut



Requête pour l'échange de jeton

La requête pour obtenir les jetons prend en paramètre :

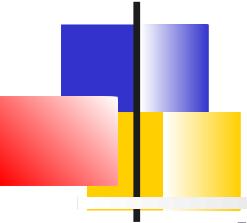
- ***grant_type*** : ***authorization_code***
- ***code*** : Le code
- ***client_id*** : Id du client
- ***redirect_uri*** : L'URI permettant au client de récupérer les jetons



Réponse jetons

La réponse JSON contient les attributs suivants :

- ***access_token*** : Jeton d'accès
- ***expires_in*** : Le jeton peut être opaque donc il y a un champ indiquant la date d'expiration
- ***refresh_token*** : Jeton d'actualisation
- ***refresh_token_expires_in*** :
- ***token_type*** : Avec Keycloak toujours Bearer
- ***id_token*** : Jeton d'identification
- ***session_state*** : ID de session de l'utilisateur dans Keycloak
- ***scope*** : Les scopes renvoyés peuvent ne pas correspondre à la requête

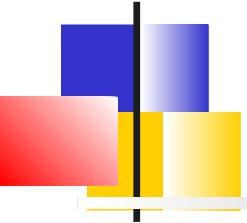


Jeton d'identification

Le jeton d'identification est par défaut un jeton JWT signé, qui suit ce format :

<En-tête>.<Charge utile>.<Signature>

L'en-tête et la charge utile sont des documents JSON encodés en Base64URL.



Revendications

exp : lorsque le jeton expire.

iat : date à laquelle le jeton a été émis.

auth_time : lorsque l'utilisateur s'est authentifié pour la dernière fois.

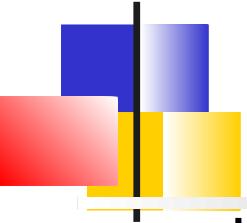
jti : L'identifiant unique pour ce jeton.

aud : l'audience du jeton, qui doit contenir la partie de confiance qui authentifie l'utilisateur.

azp : la partie à laquelle le jeton a été émis.

sub : l'identifiant unique de l'utilisateur authentifié.

Les autres revendications sont plus informatives sur l'utilisateur et peuvent être finement configurées



Rafraîchissement

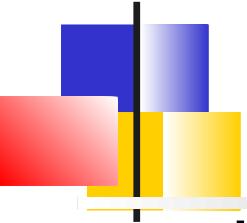
Les jetons d'identification ont une courte durée de vie afin d'atténuer le risque de fuite de jetons.

- Cela ne signifie pas que l'application doit ré-authentifier l'utilisateur

Il existe un jeton d'actualisation distinct qui peut être utilisé pour obtenir un jeton d'identification mis à jour.

- Le jeton d'actualisation a une expiration beaucoup plus longue.
- Généralement, il ne peut s'utiliser qu'une seule fois

=> Les applications peuvent mettre à jour les informations utilisateurs sans se ré-authentifier



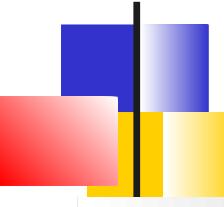
UserInfo endpoint

Le point d'accès ***UserInfo*** permet d'obtenir les informations de l'utilisateur authentifié.

- Un sous-ensemble des informations de l'ID token ne contenant que les attributs de l'utilisateur
- Les informations renvoyées peuvent être configurées

Il est accessible avec un jeton d'accès.

`http://<server>/realms/<realm-name>/protocol/openid-connect/userinfo`



PAR : Pushed Authorization Request

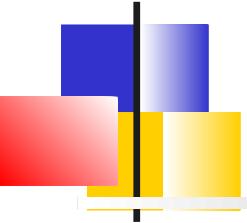
PAR (RFC 9126) (Pushed Authorization Request) est une extension du flow d'autorisation OAuth2, qui permet au client de transmettre sa requête d'autorisation directement au serveur d'autorisation via un canal sécurisé (POST HTTPS), au lieu de passer tous les paramètres dans l'URL (GET).

- 1) Le client prépare la requête d'autorisation (*client_id*, *redirect_uri*, *response_type*, *scope*, etc.)
- 2) Il envoie cette requête en POST au PAR endpoint
- 3) Keycloak répond avec un ***request_uri*** et un ***expires_in***
- 4) Le client redirige l'utilisateur vers l'URL d'autorisation standard, avec un seul paramètre *request_uri*

Contourne les inconvénients du flow standard :

- Envoi des paramètres sensibles (*scope*, *redirect_uri*, *state*, etc.) dans l'URL redirigée vers l'utilisateur
- Peut poser des problèmes de sécurité, d'intégrité, de longueur de requête, etc.

Nécessaire pour FAPI-2.0



Authentification avec OpenID

Discovery endpoint

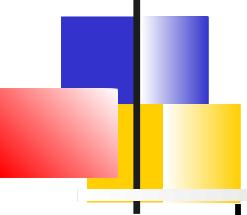
Authentification

Authentification via mTLS

CIBA Flow

Personnalisation du jeton

Logout



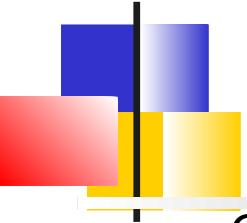
Authentification par certificat client (mTLS)

Le client s'authentifie avec un certificat TLS client, via HTTPS mutuel (mTLS).

Keycloak utilise ce certificat pour identifier et authentifier l'utilisateur sans saisie de login/mot de passe.

Avantages de mTLS

- Sécurité forte : authentification à deux facteurs implicite (possession du certificat)
- Sans mot de passe : pas de credential à stocker, pas de phishing
- Adapté aux machines ou environnements contrôlés : terminaux d'entreprise, API inter-applications
- Compatible avec un flow OAuth2 / OIDC standard si bien configuré



Inconvénients / limites de mTLS

Complexité de déploiement :

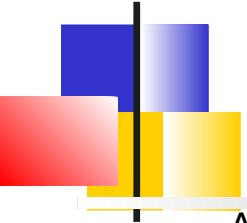
- Génération et gestion des certificats client (PKI interne)
- Configuration des reverse proxies pour le mTLS

Expérience utilisateur moins fluide :

- Nécessite parfois installation manuelle des certificats sur les terminaux

Pas adapté au grand public (ex. applications mobiles ou utilisateurs anonymes)

Pas natif dans tous les navigateurs/appareils mobiles



Authorization code Flow avec mTLS

Accès initial :

- Le client accède à Keycloak via HTTPS avec authentification mutuelle (mTLS).
- Le certificat client est présenté dès la connexion TLS.

Configuration côté Keycloak :

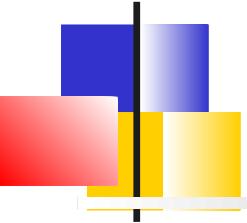
- Le certificat client est requis ou demandé (`HTTPS_CLIENT_AUTH=request|required`)
- La validation du certificat est assurée par Keycloak ou par un reverse proxy TLS (ex : NGINX en mTLS).

Authentification de l'utilisateur :

- Keycloak utilise un authenticator SPI X.509 pour mapper le certificat à un utilisateur : mapping via : subjectDN, issuerDN, serialNumber, SAN, etc.
- L'utilisateur est automatiquement connecté, sans saisie de login/mot de passe.

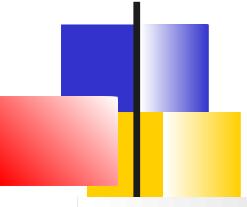
Mise en œuvre :

- Ajouter l'authenticator X.509 dans le flow “browser”.
- Configurer un proxy TLS si besoin (pour extraire le certificat).
- Définir les règles de mapping certificat → user dans Keycloak.



Authentification avec OpenID

Discovery endpoint
Authentification
Authentification via mTLS
CIBA Flow
Personnalisation du jeton
Logout

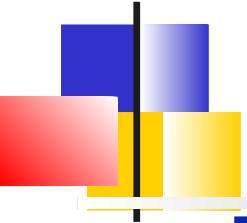


CIBA Flow

Client-Initiated Backchannel Authentication (CIBA) est une extension d'OpenID Connect pour l'authentification asynchrone sans redirection navigateur. Idéale pour les appareils sans interface utilisateur (IoT, banques, assistants vocaux...).

Principe :

- Le client n'ouvre pas de navigateur.
- Il initie l'authentification côté serveur (via un backchannel).
- Le fournisseur d'identité notifie l'utilisateur sur un autre canal (ex : mobile, push).
- L'utilisateur accepte/refuse, et le jeton est délivré ensuite.



Terminologie CIBA

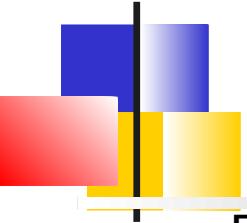
Backchannel Authentication Endpoint : Point d'entrée CIBA du serveur d'authentification

login_hint / id_token_hint : Identifie le user à authentifier

Binding Message : Message à afficher à l'utilisateur

Notification Endpoint (optionnel) : endpoint de callback du client pour recevoir le résultat de l'authentification

Poll / Ping / Push : Méthodes de récupération du token après autorisation



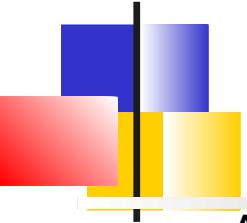
Flow CIBA

Flow pour l'obtention du jeton

- 1) Le client envoie une requête d'authentification backchannel au serveur, avec un *login_hint* ou *id_token_hint*, et optionnellement un *notification_endpoint*.
- 2) Le serveur notifie le client via le *notification_endpoint*, quand l'utilisateur a terminé l'authentification, que ce soit un succès ou un échec.
- 3) Le client utilise cette notification et l'*auth_req_id* qu'il a reçu à l'étape 1 pour récupérer un token auprès du token endpoint.

Flow pour l'authentification

- 1) Lors d'une requête d'authentication, le serveur d'autorisation notifie un service
- 2) Le service acquitte puis demande à authentifier l'utilisateur
- 3) Il informe le serveur du résultat de l'authentification. Si elle réussie, le jeton est disponible pendant x temps



Support dans Keycloak

Activation par client (Enable CIBA)

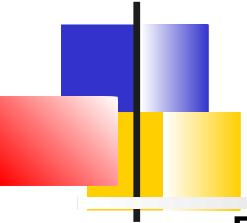
Configuration générale *Authentication → CIBA Policy*, peut être surchargé par client (pas supporté par l'UI par contre) :

- Mode de livraison du jeton : Seulement ping
- Expiration de la demande authentification (120s par défaut)
- Intervalle de polling
- Comment identifier l'utilisateur : Seulement login_hint

Canal d'authentification implémenté sous forme de SPI

- Soit on implémente son propre canal
- Soit on utilise le SPI prédéfini : **HTTP Authentication Channel Provider**

Pas de support pour le notification endpoint dynamique



HTTP Authentication Chanel Manager

Doit être activé en temps que SPI :

```
kc .[sh|bat] start --spi-ciba-auth-channel-ciba-http-auth-
channel-http-authentication-channel-uri=
https://backend.internal.example.com
```

Après une demande d'authentification

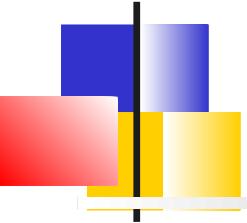
POST Keycloak avec un jeton

Le backend répond avec un 201 et conserve le jeton

Lorsque l'utilisateur est authentifié, le backend ou le client renvoie une requête POST vers :

/realms/[realm]/protocol/openid-connect/ext/ciba/auth/callback
avec :

- Le jeton initial
- Le résultat de l'authentification



Authentification avec OpenID

Discovery endpoint

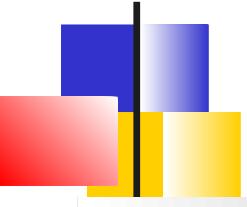
Authentification

Authentification via mTLS

CIBA Flow

Personnalisation du jeton

Logout



Introduction

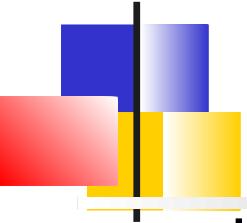
Les différents clients d'une application n'ont pas nécessairement besoin des mêmes informations dans les jetons.

Dans OAuth2 / OIDC, les *scopes* permettent de définir les droits ou informations qu'un client peut demander (ex. openid, email, profile).

Keycloak implémente ce mécanisme via 2 composants clés :

- **Client Scopes** : ensembles configurables regroupant des paramètres (notamment des mappers) associés à un ou plusieurs scopes OAuth2/OIDC.
Ils peuvent être automatiquement attachés ou activés explicitement lors de l'authentification.
- **Protocol Mappers** : définissent les informations (claims) ajoutées dans les tokens.

En combinant client scopes et mappers, Keycloak adapte dynamiquement les jetons en fonction des scopes OAuth2 demandés.



Protocol Mappers

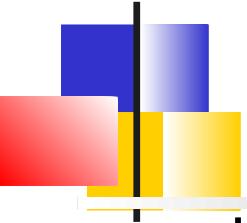
Les **Mappers** permettent de personnaliser les revendications des jetons fournies à un client:

- Informations statiques,
- Attributs d'utilisateur
- Rôles.

Ils s'appliquent aux tokens ID, tokens d'accès, et à l'endpoint *userinfo* selon la configuration

Keycloak fournit des mappers prédéfinis.

Les mappers peuvent être définis au niveau d'un client ou d'un client scope



Client Scopes

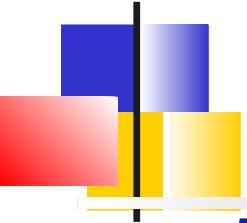
Les ***client scopes*** permettent de définir des configurations client qui pourront être mutualisées entre différents clients.

Cela englobe principalement

- Les mappers : Résolution des claims
- Le *Scope ou Role Scope mapping* : les rôles de l'utilisateur auxquels le client a accès

Les *client scopes* prennent en charge le paramètre *scope* d'OAuth 2.

C'est donc le moyen d'adapter les revendications et en particulier les rôles en fonction des exigences de l'application.

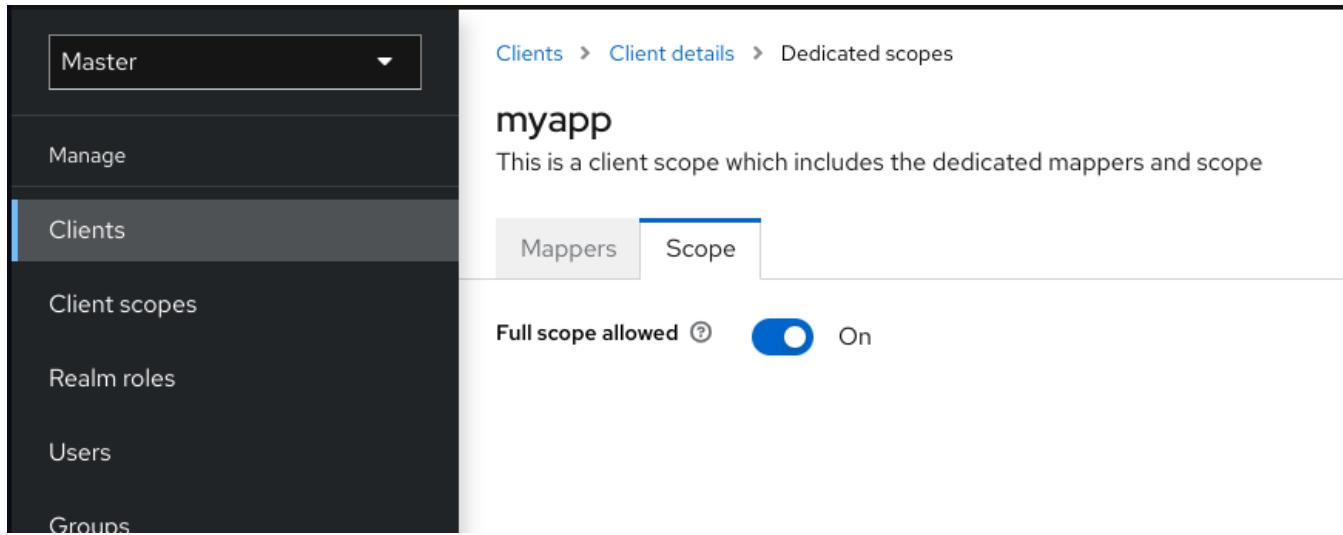


Role scope mapping

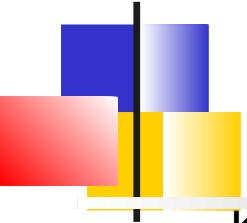
Role scope Mapping permet de limiter les rôles indiqués dans un jeton d'accès.

Le jeton d'accès reçu par un client ne contient que les rôles explicitement spécifiés pour le scope du client.

Si non défini, chaque client obtient tous les rôles de l'utilisateur.



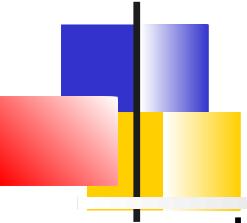
The screenshot shows the Keycloak management interface. On the left, a sidebar menu includes 'Master', 'Manage', 'Clients', 'Client scopes', 'Realm roles', 'Users', and 'Groups'. The 'Clients' item is selected. The main content area shows the 'Clients' section with 'myapp' selected. The URL in the browser bar is 'Clients > Client details > Dedicated scopes'. Below the client name, it says 'This is a client scope which includes the dedicated mappers and scope'. There are two tabs: 'Mappers' (disabled) and 'Scope'. A toggle switch labeled 'Full scope allowed' is set to 'On'. The status bar at the bottom indicates 'On'.



Client Scopes

Keycloak propose des client scopes prédéfinis pour OpenID :

- **roles** : Pas défini par OpenID Connect et pas ajoutée automatiquement.
Il comporte des mappeurs utilisés pour ajouter les rôles de l'utilisateur au jeton d'accès et ajouter des audiences pour les clients qui ont au moins un rôle client.
- **web-origins** : Pas défini dans OpenID Connect et pas ajoutée automatiquement.
Utilisé pour ajouter des origines Web autorisées dans le jeton d'accès. (allowed-origins)
- **microprofile-jwt** : Gère les revendications définies dans la spécification MicroProfile/JWT.
- **offline_access** : Spécification OpenID Connect. Permet d'obtenir un *offline refresh token*. Permet aux applis de rester inactive longtemps
- **profile, email, address, phone** : Défini dans OpenID Connect : des mappers mais pas de rôle mapping



Liens entre Client et Client Scopes

L'association entre client et Client scopes s'effectue dans l'onglet

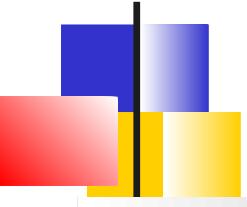
Client → Client Scopes

2 types de liaisons :

- ***Default Client Scopes*** : Le client hérite des mappeurs et des mapping de rôles dans tous les cas.
- ***Optional Client Scopes*** : Le client hérite des mappeurs et des mapping de rôles seulement si le paramètre scope de la requête OpenID a mentionné ce scope.

De plus, chaque client a un client-scope dédié automatiquement appliqué quelque soit le scope oauth2 demandé.

Settings	Keys	Credentials	Roles	Client scopes	Sessions	Advanced
Setup	Evaluate					
Name	Search by name	Add client scope	Change type to		Refresh	
<input type="checkbox"/> ciba-client-dedicated	none	Dedicated scope and mappers for this client				
<input type="checkbox"/> acr	Default	OpenID Connect scope for add acr (authentication context class reference) to the token				
<input type="checkbox"/> address	Optional	OpenID Connect built-in scope: address				
<input type="checkbox"/> email	Default	OpenID Connect built-in scope: email				
<input type="checkbox"/> microprofile-jwt	Optional	Microprofile - JWT built-in scope				
<input type="checkbox"/> offline_access	Optional	OpenID Connect built-in scope: offline_access				
<input type="checkbox"/> phone	Optional	OpenID Connect built-in scope: phone				

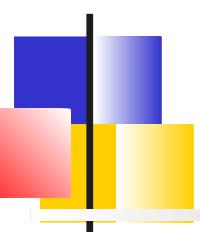


Évaluer les client scopes

Il est possible d'évaluer les mappeurs effectifs et les mapping de rôle qui seront générés.

Client Scopes → Evaluate

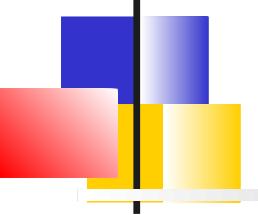
Choisir ensuite les scopes optionnels éventuels



Permissions des client scopes

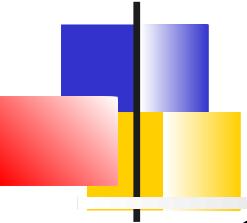
A l'émission d'un jeton, les client scopes ne sont appliqués que si l'utilisateur y est autorisé.

- Lorsqu'aucun rôle mapping n'est défini, Tous les utilisateurs peuvent utiliser tous les scopes.
- Quand un client scope contient des mappings de rôles, seuls les utilisateurs possédant au moins un de ces rôles peuvent utiliser ce scope.
=> Il doit y avoir une intersection entre les rôles d'utilisateur et les rôles du scope client.



Authentification avec OpenID

Discovery endpoint
Authentification
Authentification via mTLS
CIBA Flow
Personnalisation du jeton
Logout



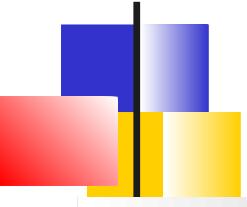
Introduction

Gérer la déconnexion dans une expérience SSO peut être assez difficile, surtout si on souhaite une déconnexion instantanée de toutes les applications qu'un utilisateur utilise.

Un logout peut être initié par un timeout qui expire ou par une action manuelle de l'utilisateur

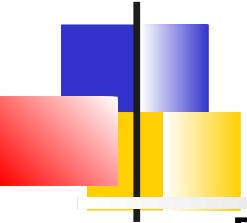
En fonction des différents types de clients, la déconnexion est répercutée de façon différentes

La déconnexion doit également provoquer l'invalidation des jetons actifs de l'utilisateur



Mécanisme du logout

1. Une déconnexion peut être initiée par l'utilisateur en cliquant sur un bouton de déconnexion dans une application.
2. L'application nettoie son propre contexte de sécurité et redirige l'utilisateur vers le `end_session_endpoint` de Keycloak pour le prévenir avec comme paramètres :
 - **`id_token_hint`** : un jeton d'identification émis précédemment.
 - **`post_logout_redirect_uri`** : Si le client souhaite que Keycloak redirige vers une URI après la déconnexion.
 - **`state`** : Maintient l'état entre la demande de déconnexion et la redirection.
 - **`ui_locales`** : Optionnel locale devant être utilisée pour l'écran de connexion.
3. Keycloak invalide les sessions en cours. Tous les jetons de l'utilisateur deviennent invalides.
4. Si configuré, Keycloak informe les autres clients de la déconnexion en utilisant le canal **`front`** ou **`back`**
5. L'utilisateur est redirigé vers `post_logout_redirect_uri`

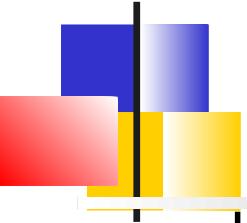


Découverte de la déconnexion

Plusieurs mécanismes peuvent être mis en œuvre, en fonction du type de client et du niveau de sécurité attendu :

- Expiration naturelle ou échec de rafraîchissement des tokens
- OpenID Connect Session Management
- OIDC Back-Channel Logout
- OIDC Front-Channel Logout

Les méthodes 2 à 4 permettent une propagation active de la déconnexion SSO à plusieurs applications.



Détection par expiration

L'application se rend compte de la déconnexion quand un token n'est plus utilisable.

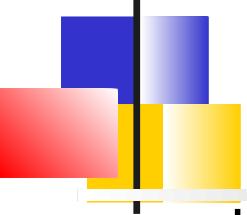
- Les tokens d'accès et ID tokens ont une durée de vie courte (ex. 5 min).

L'application essaie de rafraîchir le token via le refresh_token.

Si ce dernier est révoqué ou expiré, l'utilisateur doit se reconnecter.

Avantages : simple, ne nécessite aucune configuration spéciale.

Limite : la déconnexion n'est pas immédiate pour les autres clients.



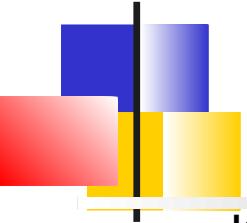
OIDC Session Management (Front-Channel Polling)

L'application surveille l'état de la session SSO via une iframe.

- Implémenté via un iframe embarquant une page spéciale de Keycloak.
/realms/\${realm}/protocol/openid-connect/login-status-iframe.html
- L'application effectue un polling régulier pour détecter la fin de session.
- Si Keycloak détecte que la session est fermée, l'app déclenche un logout local.

Avantages : pas besoin de gérer un serveur côté application.

Limites : dépend du navigateur, nécessite le support de l'iframe, consommation réseau (poll).



OIDC Back-Channel Logout

Keycloak informe directement les backends des applications.

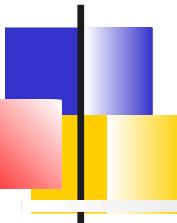
- Chaque client déclare une URL de logout back-channel.
- Lorsqu'un utilisateur se déconnecte, Keycloak envoie une requête HTTP POST signée (logout token) à cette URL.
- L'application invalide la session côté serveur.

Avantages :

- Sécurisé, indépendant du navigateur
- Fonctionne même si l'utilisateur ferme le navigateur

Limites :

- Requiert un backend réactif
- Doit gérer la signature du logout token



OIDC Front-Channel Logout

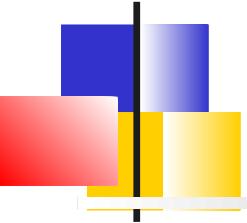
Propagation du logout via des iframes client dans la page de logout Keycloak.

- Chaque client enregistre une URI de logout front-channel.
- Lors du logout, Keycloak affiche une page contenant une iframe par client.
- Ces iframes appellent la page de logout du client pour qu'il nettoie la session.

Avantages : pas de backend requis.

Limites :

- Très sensible aux Content Security Policy (CSP) ou aux navigateurs
- Fragile, nécessite une configuration soignée

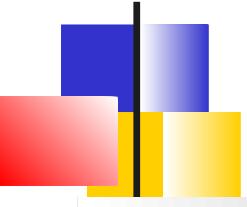


Autoriser les accès avec oAuth2

Jeton d'accès et consentement

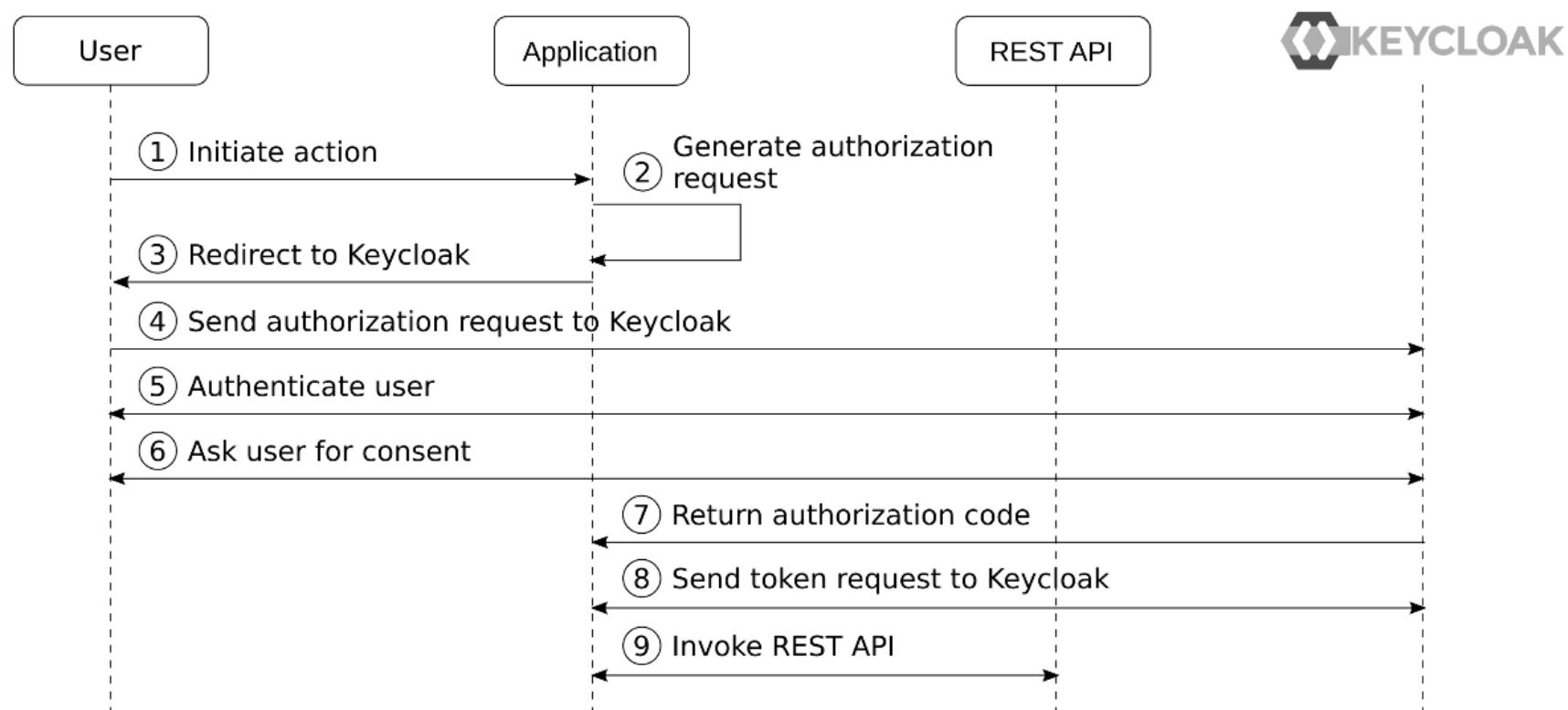
Limitations des accès

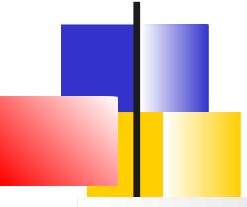
Validation du jeton



Obtenir un jeton d'accès

L'**Authorization-flow** est le moyen le plus courant d'obtenir un jeton d'accès. Ce flux inclut normalement le consentement de l'utilisateur

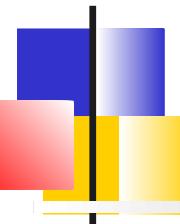




Contenu du jeton d'accès

Les attributs les plus importants du jeton d'accès :

- ***aud*** (audience) : Liste de services auxquels ce jeton est destiné à être envoyé.
- ***realm_access*** : Liste de rôles realm auxquels le jeton donne accès : intersection des rôles attribués à un utilisateur et des rôles autorisés auxquels une application est autorisée d'accéder.
- ***resource_access*** : Liste de rôles client auxquels le jeton donne accès.
- ***scope*** : Scope inclus dans le jeton d'accès.



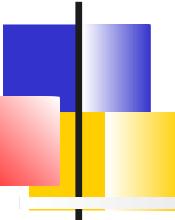
Consentement de l'utilisateur

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

Le type de privilèges d'accès demandé par l'application est contrôlé par les scopes demandés par l'application.

- Les scopes peuvent être demandés *progressivement* par l'application

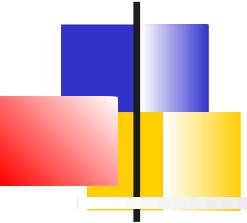
Les consentements de l'utilisateur sont conservés par Keycloak. Un utilisateur peut supprimer ses consentements en accédant à Keycloak



Application interne / externe

Pour une application interne, il n'est pas nécessaire d'obtenir le consentement de l'utilisateur

- Cette application « corporate » de confiance doit être accessibles pour les utilisateurs de l'entreprise
- => C'est la valeur par défaut car Keycloak est dédié à l'entreprise

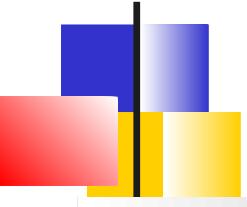


Autoriser les accès avec oAuth2

Jeton d'accès et consentement

Limitations des accès

Validation du jeton

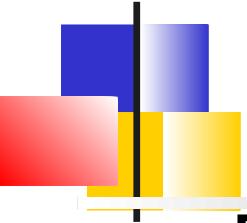


Limiter les accès du jeton

Les accès du jeton peuvent être limités selon 3 axes principaux :

- **Audience** : permet de lister les fournisseurs de ressources qui doivent accepter un jeton d'accès.
- **Rôles** : Les ACLs sur les ressources sont exprimées en fonction des rôles utilisateur.
- **Scopes¹** : Les ACLs sur les ressources sont exprimées en fonction des scopes clients.

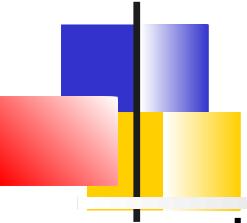
¹. Mécanismes par défaut de oAuth2, les scopes oAuth2 sont mappés sur les client scopes.



Configuration des audiences

Dans Keycloak, il existe 2 manières différentes d'inclure un client dans l'audience.

- Manuellement en ajoutant un client spécifique à l'audience à l'aide d'un mapper.
On peut par exemple utiliser le scope client dédié au client toujours présent dans le jeton
- Automatiquement, si le client a une scope sur un rôle client d'un autre client.

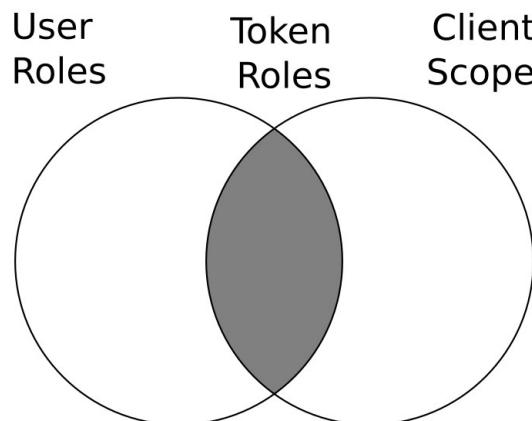


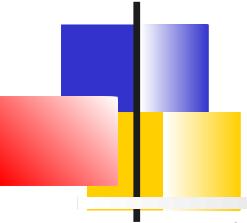
Configuration des rôles

Un utilisateur est affecté à un certain nombre de rôles.

Un client, n'a pas de rôles directement assignés, mais a un *scope* sur un ensemble de rôles, qui contrôle quels rôles peuvent être inclus dans les jetons envoyés au client

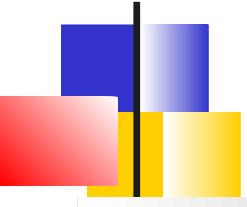
=> les rôles inclus dans les jetons sont l'intersection entre les rôles d'un utilisateur et les rôles qu'un client est autorisé à utiliser





Configuration des scopes du client

- ✓ Un client a une scope sur les rôles. Les rôles autorisés pour ce client. Par défaut « Full scope » :
Ceci est configuré via l'onglet
Client → Client Scopes → Client Dedicated → Scope du client.
- ✓ Un client peut accéder à un ou plusieurs scopes client.
Ceci est configuré via l'onglet ***Client → Client Scopes*** du client.
 - ✓ Les client Scopes sont par défaut ou optionnel
- ✓ Un « scope client » peut également avoir un scope sur les rôles.
Lorsqu'un client a accès à ce « scope client », il obtient le scope sur les rôles correspondant.
Client Scopes → Scope du client scope.



ACLs sur les scopes

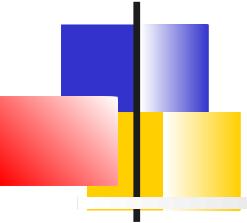
Le mécanisme par défaut dans OAuth 2.0 pour limiter les autorisations pour un jeton d'accès passe directement par les scopes.

Dans Keycloak, un scope de OAuth 2.0 est mappé à une scope client.

- Si l'on souhaite uniquement disposer d'une scope pour fixer des ACLs sur un serveur de ressource, il suffit de définir un client scope avec aucun mapper ni aucun rôle associé .

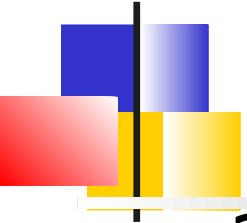
Ce type de scopes sont généralement dédiés à une application de l'entreprise. Leur nom peut être préfixé par le nom ou l'URL du service

- albums:view, <https://api.acme.org/api/albums.view>,
<https://www.googleapis.com/auth/calendar.events>, ...



Autoriser les accès avec oAuth2

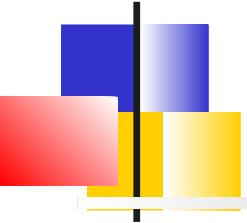
Jeton d'accès et consentement
Limitations des accès
Validation du jeton



Validation du jeton

2 choix pour valider un jeton d'accès :

- Appeler le endpoint d'introspection
Respect pur du standard oAuth2 (les jetons sont supposés opaques)
Mais ajoute une requête supplémentaire
- Vérifier directement le jeton grâce à JWT
JWT permet de parser les informations directement
La signature permet de s'assurer de l'authenticité du jeton via une clé publique

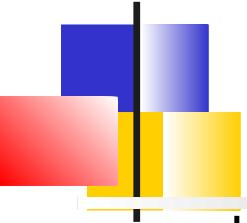


Sécurisation des différents types d'application

Application web

Application native ou mobile

REST APIs et services



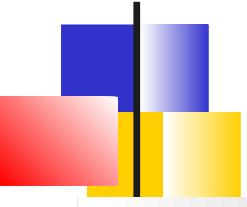
Types d'application web

La sécurisation d'une application web dépend de l'architecture de l'application, on peut distinguer :

- Traditionnelle : L'application Web s'exécute entièrement à l'intérieur d'un serveur « backend ».
- SPA avec API dédiée : L'application s'exécute dans le navigateur et appelle uniquement une API dédiée sous le même domaine.
- SPA avec API intermédiaire : L'application SPA appelle des API externes via une API intermédiaire hébergée sous le même domaine
- SPA avec API externe : L'application SPA appelle des API hébergées sous un domaine différent¹.

Quelque soit l'architecture le meilleur mécanisme d'authentification d'un utilisateur reste l'*Authorization Code Flow*

1. Ex : SPA déployé sous node.js et backend déployé sous un autre serveur



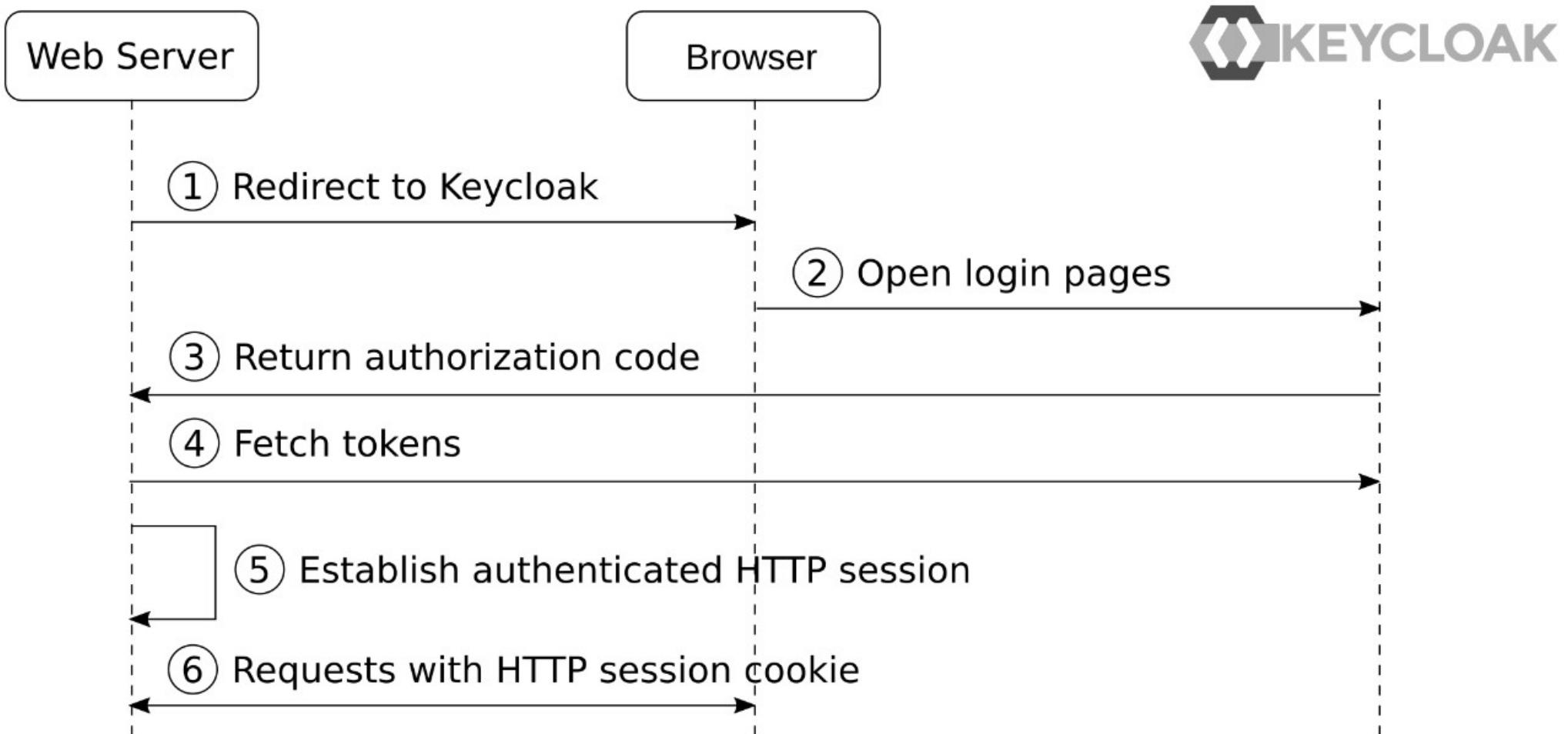
Application traditionnelle

Avec ce type d'application, seul le jeton d'identification est utilisé pour établir une session HTTP, il contient les informations nécessaires pour les ACLs de l'application.

Recommandations :

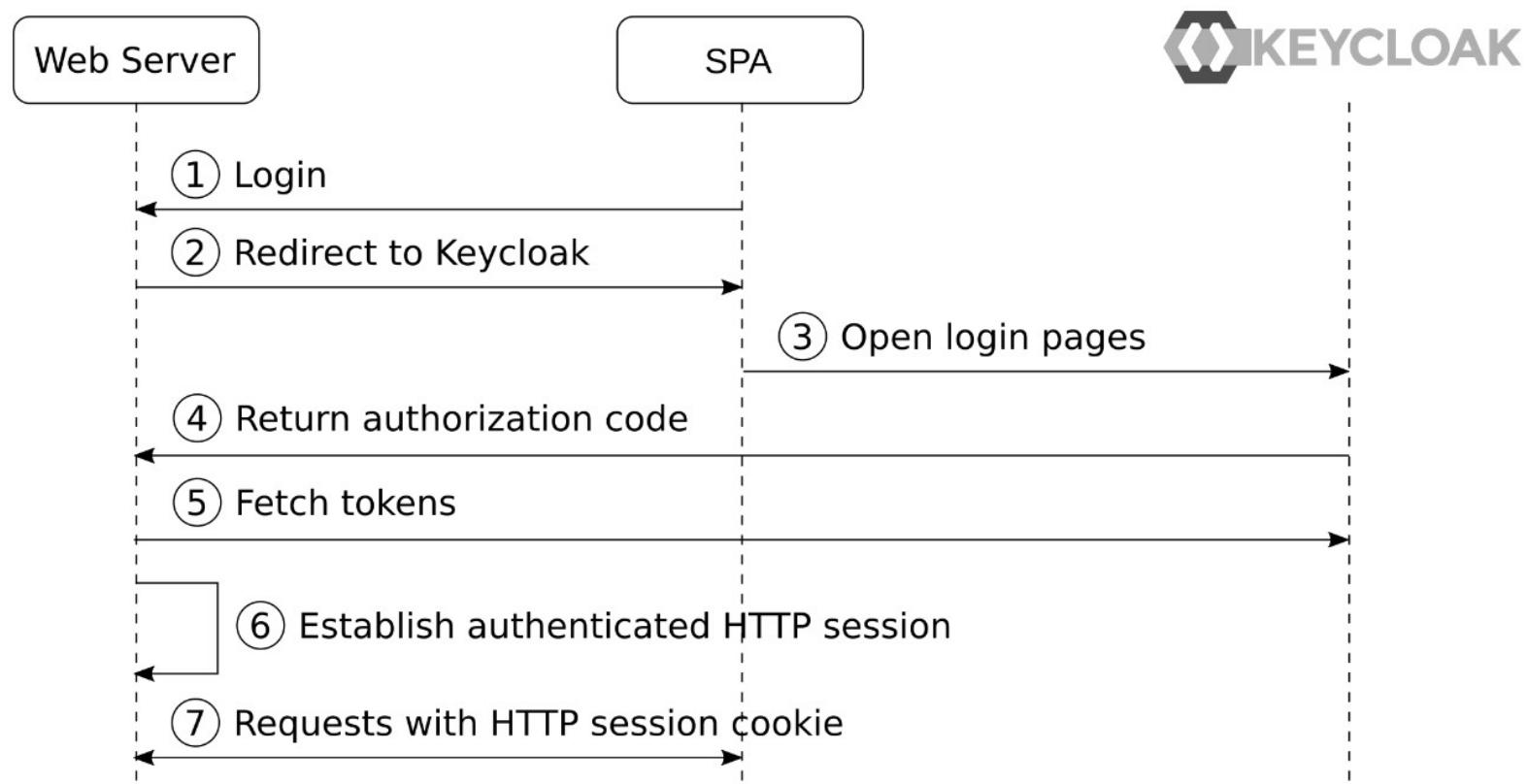
- Enregistrer un client confidentiel
Pour obtenir le jeton, il faut alors le code + le secret du client stocké dans le back-end
- On configure généralement *PKCE* pour plus de sécurité
- Donner des URLs de redirection strictes

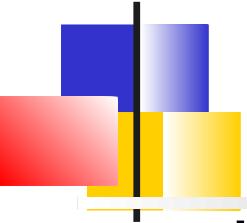
Application traditionnelle



SPA avec API dédiée

Identique au cas d'une appli web traditionnelle. Le secret est conservé côté backend





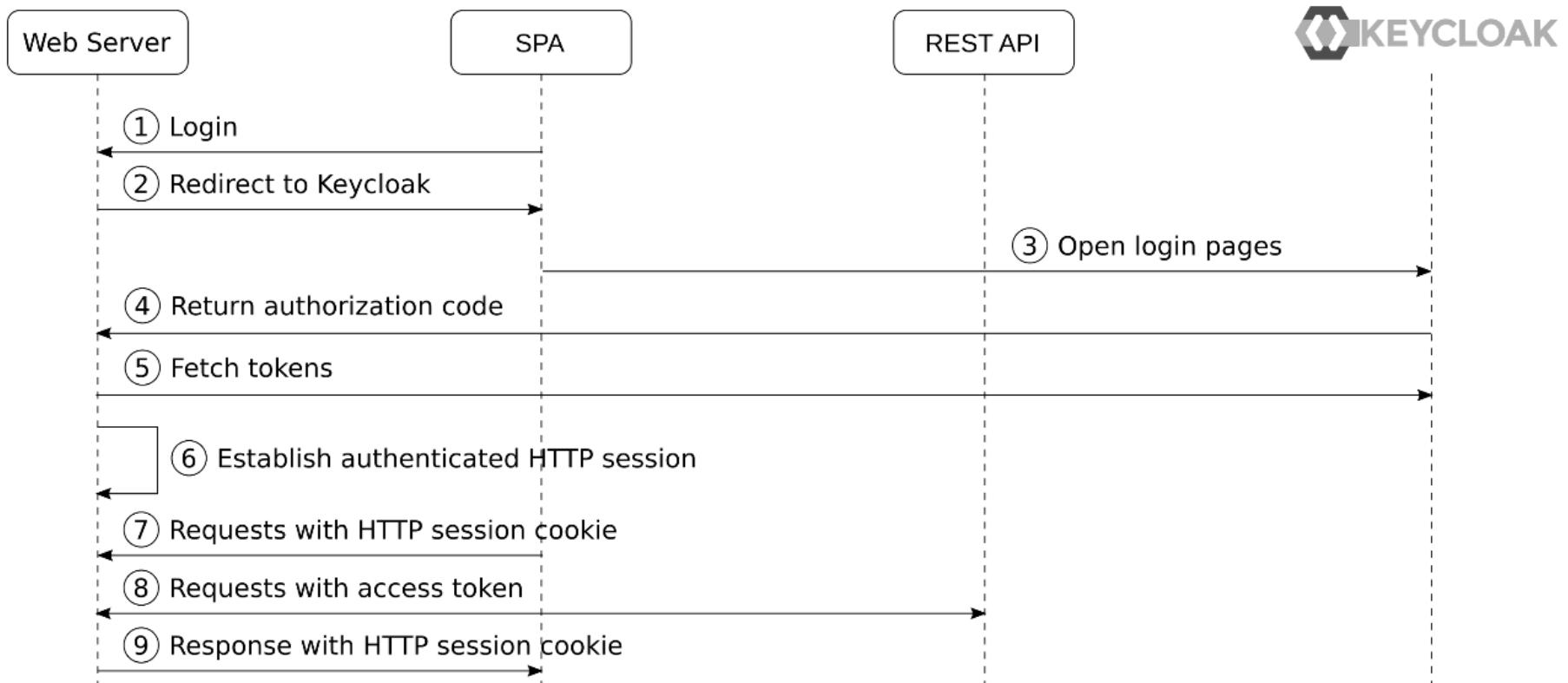
SPA avec API intermédiaire dédiée

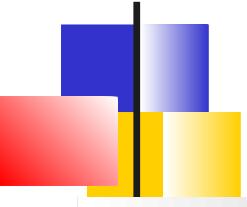
Le moyen le plus sûr d'invoquer des API externes à partir d'un SPA consiste à utiliser une API intermédiaire hébergée sur le même domaine que le SPA¹.

- Possibilité d'utiliser un client confidentiel et les jetons ne sont pas directement accessibles dans le navigateur.
- Pas besoin de mettre en place du CORS

Les appels vers les APIs externes utilisent le jeton d'accès

SPA avec API intermédiaire dédiée

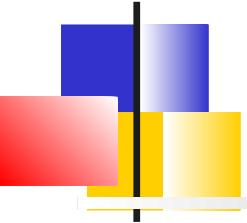




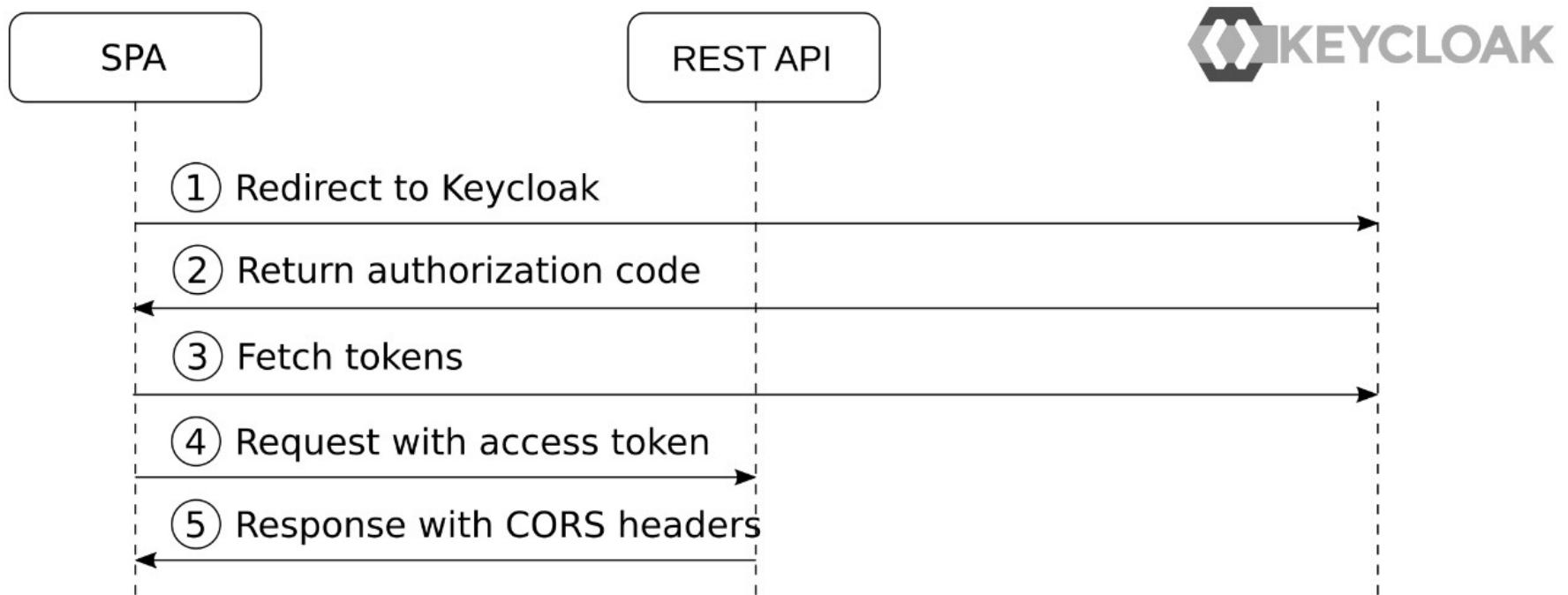
SPA avec API externe

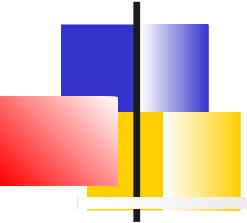
Le moyen le plus simple est d'effectuer le flux *authorization-code* depuis le SPA avec un client public enregistré dans Keycloak.

- L'approche est moins sécurisée car les jetons, y compris le jeton d'actualisation, sont exposés directement au navigateur.
 - => Avoir une courte expiration pour le jeton d'actualisation
 - => Révoquer les jetons d'actualisation après leur 1ère utilisation.
 - => Utiliser l'extension PKCE
 - => Stocker les jetons dans l'état de la fenêtre ou la session de stockage HTML5
 - => Protéger le SPA contre le Cross Script Scripting (XSS)¹



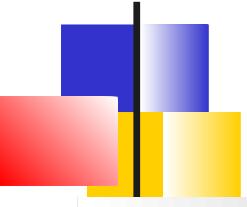
SPA avec API externe





Sécurisation des différents types d'application

Application web
Application native ou mobile
REST APIs et services



Introduction

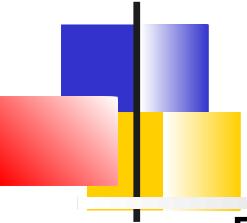
Sécuriser une application web avec Keycloak est plus simple que de sécuriser une application native ou mobile.

- Les pages de connexion étant affichées dans le navigateur.

On pourrait être tenté d'implémenter une page de connexion dans l'application et utiliser le *password grant flow*, mais mauvaise idée :

- L'application ne doit pas avoir accès aux mots de passe
- Certaines fonctionnalités de keycloak ne seraient plus disponibles comme le MFA, le SSO ou la gestion de session

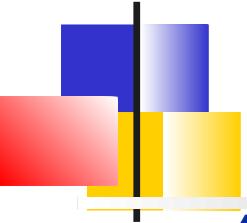
=> Le flow recommandé est donc *Authorization Code flow* avec l'extension PKCE



Authentification

Dans une application mobile ou native, l'utilisateur doit toujours s'authentifier via un navigateur :

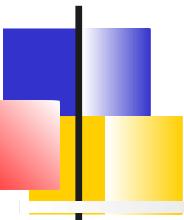
- Utilisez une vue Web intégrée dans l'application (à éviter)
 - Facile à intégrer, mais peu sécurisée
 - Risque d'interception des credentials
 - Pas de SSO possible (cookies non partagés entre apps)
- Navigateur embarqué sécurisé (in-app browser)
 - Exemple : Custom Tabs (Android), SFSafariViewController (iOS)
 - Bénéficie du SSO du navigateur système
 - Recommandé par les bonnes pratiques Oauth/OIDC
- Navigateur externe (user-agent système¹):
 - Très sûr (authentification hors de l'app)
 - Mais moins fluide : bascule visible vers le navigateur



Authorization Code avec PKCE

Authorization Code avec PKCE est aujourd’hui le standard recommandé pour les applications mobiles ou desktop.

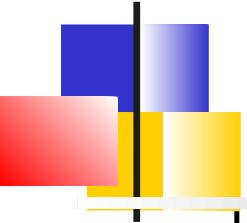
- 1) L’utilisateur clique sur “Se connecter” dans l’application.
- 2) L’application ouvre un navigateur système ou sécurisé (ex : Custom Tabs sur Android).
- 3) Keycloak affiche la page de connexion.
- 4) L’utilisateur s’authentifie.
- 5) Keycloak redirige vers l’application via un mécanisme compatible (URI personnalisée, loopback...).
- 6) L’application reçoit un code d’autorisation.
- 7) Elle échange ce code contre des tokens, en ajoutant un code PKCE pour sécuriser la requête.
- 8) Keycloak valide la requête et renvoie les tokens.



Renvoi du code d'autorisation

Pour renvoyer le code d'autorisation à l'application, 4 approches basées sur des URI de redirection définies par OAuth 2.0 :

- **Claimed HTTPS scheme** : App Links (Android) / Universal Links (iOS) : l'app revendique un domaine HTTPS et reçoit la redirection automatiquement via l'OS.
=> L'OS ouvre l'URI dans l'application au lieu du navigateur système.
- **Schéma d'URI personnalisé** : Un schéma d'URI personnalisé est enregistré avec l'application. Lorsque Keycloak redirige vers ce schéma d'URI, la requête est envoyée à l'application.
Le schéma d'URI correspond à l'inverse d'un domaine du fournisseur de l'application. Par exemple :
`org.acme.app://oauth2/provider-name`
Plus simple à configurer mais moins sécurisé
- **Interface de loopback** : l'application ouvre un serveur Web temporaire sur l'interface de loopback, l'URI de redirection enregistré est
`http://127.0.0.1/oauth2/provider-name`
Recommandé pour les applications desktop
- **URI de redirection spéciale**: En utilisant `urn:ietf:wg:oauth:2.0:oob`, le code d'autorisation est affiché par Keycloak, l'utilisateur peut alors le copier/coller dans l'application.
Déconseillé : méthode obsolète et non recommandée par les spécifications récentes d'OAuth.



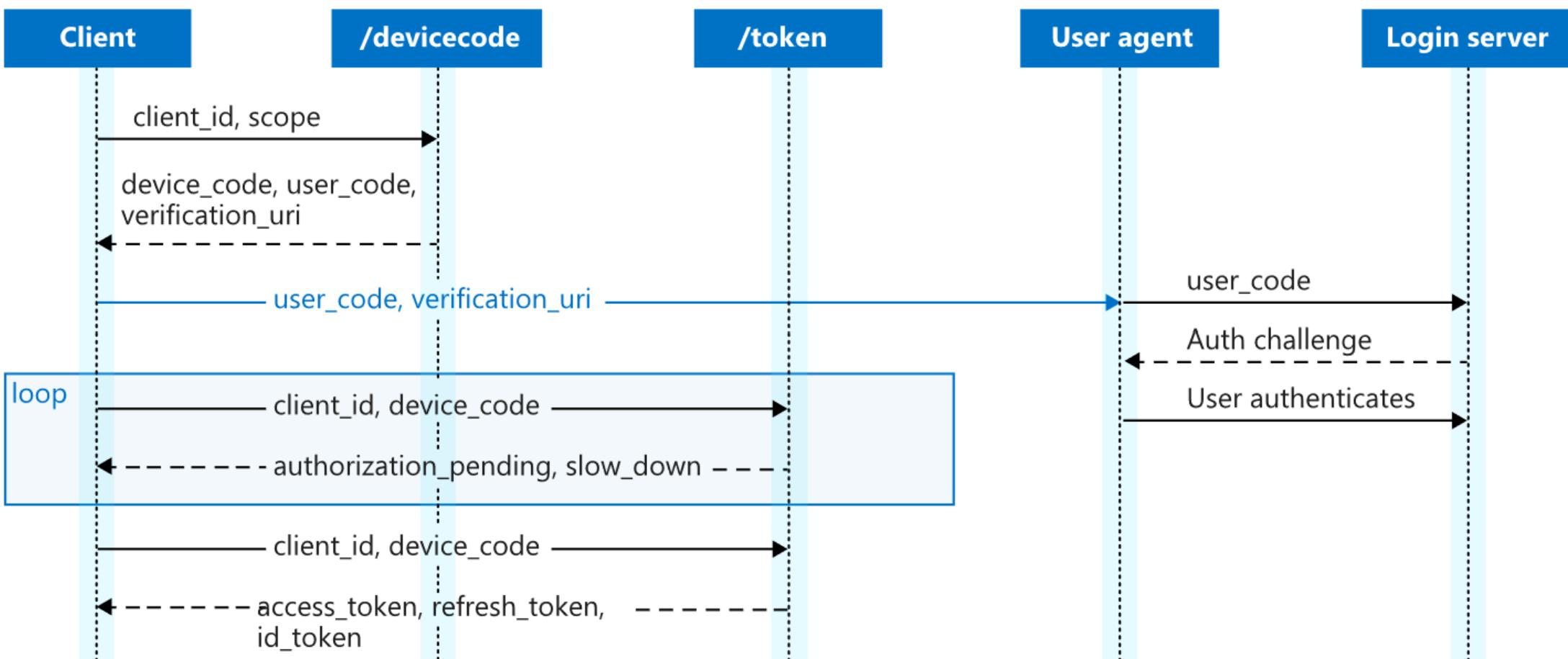
Device Code Flow

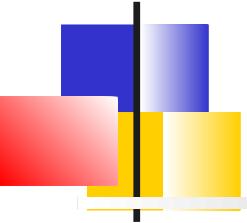
Le **Device Code Flow** permet à une application sans navigateur intégré (ex. Smart TV, CLI) d'obtenir des jetons via un second appareil avec navigateur.

- 1) L'application effectue un POST sur le *device_authorization_endpoint*.
Keycloak répond avec un **code utilisateur** et un **code device**.
- 2) L'utilisateur saisit le code utilisateur sur un endpoint de keycloak (*verification_uri*) en utilisant un autre device avec navigateur
- 3) Après avoir entré le code, l'utilisateur s'authentifie et consent
- 4) L'application est ensuite capable de récupérer les jetons en utilisant son device code

Ce flow n'utilise pas de redirection et ne supporte pas le SSO.

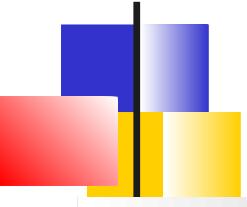
Device Code Flow





Sécurisation des différents types d'application

Application web
Application native ou mobile
REST APIs et services



Introduction

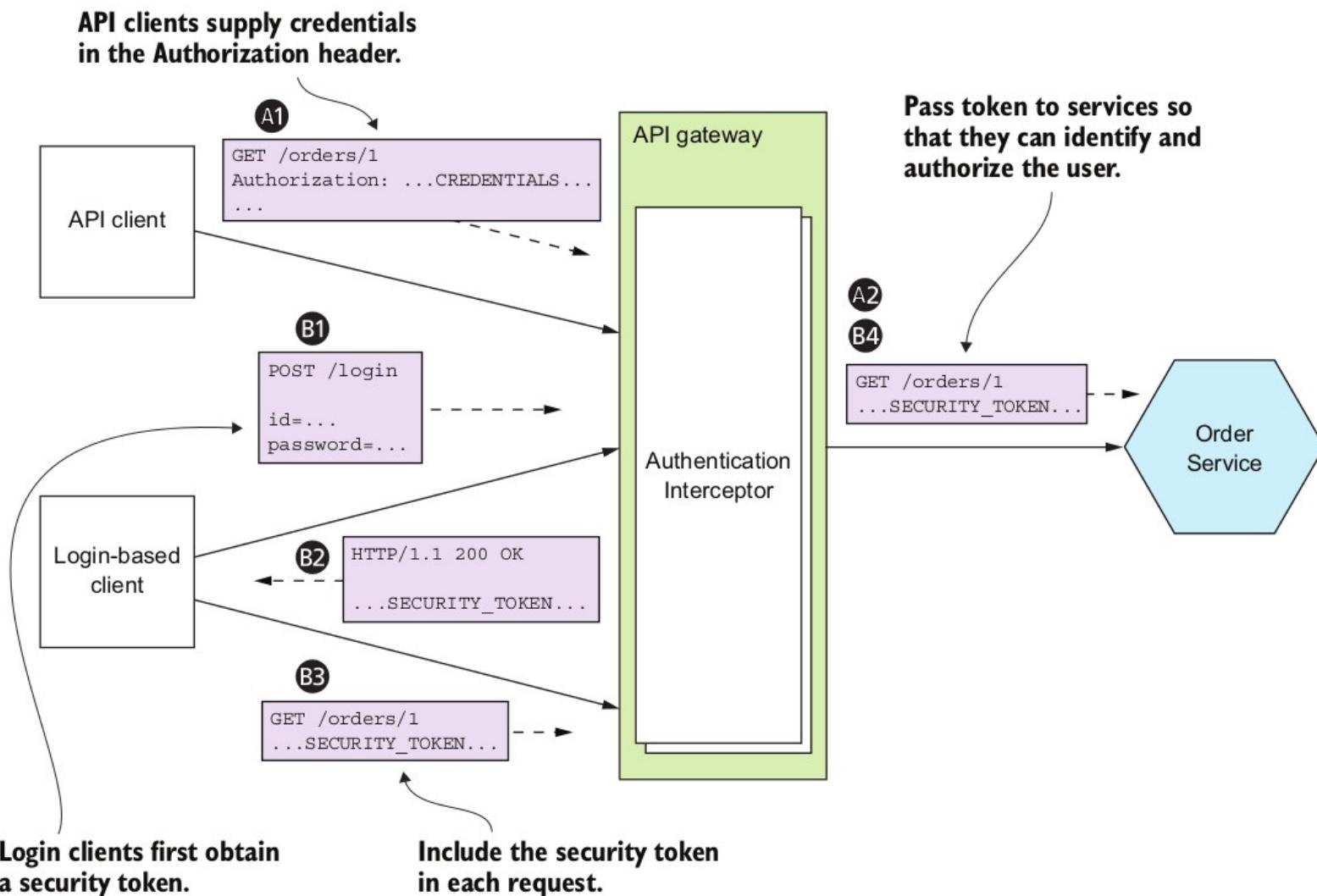
Lorsqu'une application souhaite invoquer une API protégée par oAuth2, elle obtient d'abord un jeton d'accès de Keycloak, puis inclut le jeton d'accès dans l'en-tête d'autorisation.

Lors des architecture micro-services, 2 alternatives sont possibles :

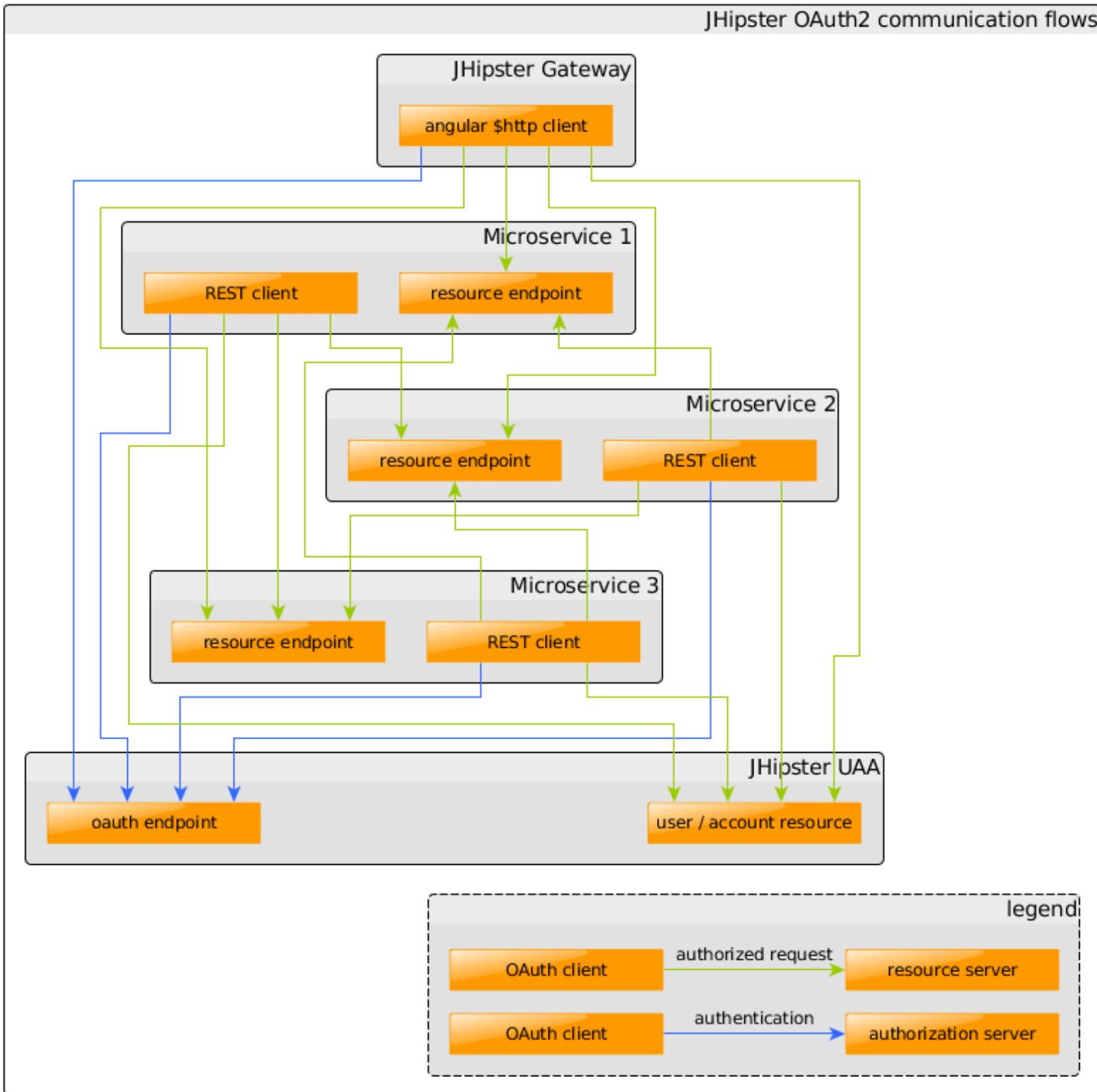
- Propagation de jeton, le même jeton est propagé lors des appels inter-services. Tous les micro-services partagent le même contexte de sécurité
- Chaque micro-service utilise son propre jeton obtenu avec un *Client Credentials grant*. Chaque interaction a alors son propre contexte de sécurité

Access Token Pattern

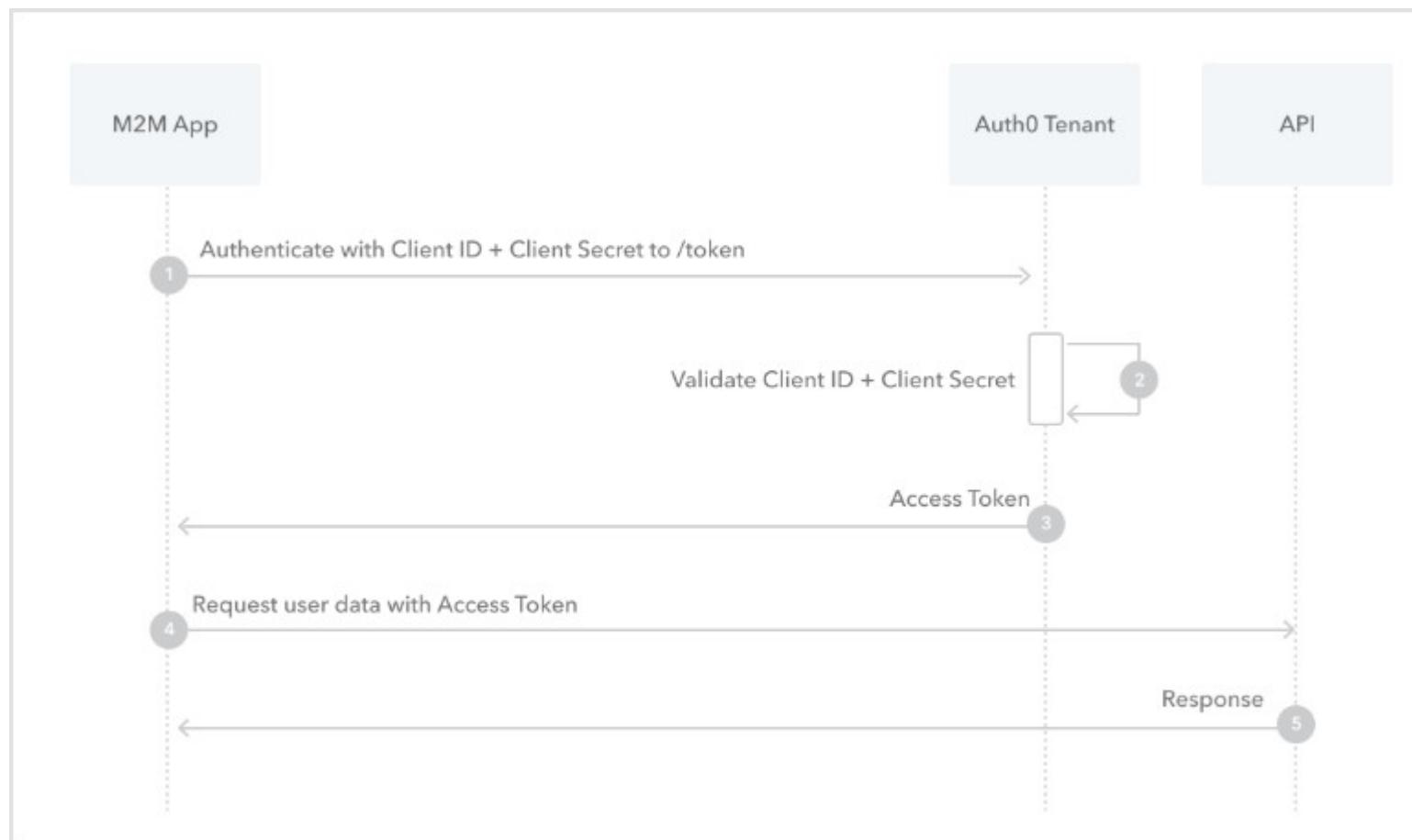
La gateway relaie le jeton d'accès

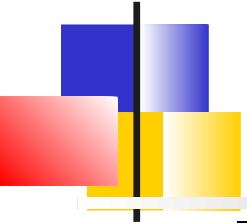


JHipster OAuth2 communication flows



Client Credentials Flow avec secret





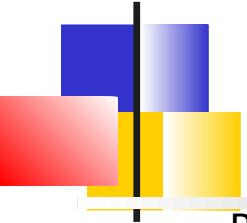
Client credentials avec certificat X509

Dans ce cas, une application backend obtient un jeton via le flow client credentials

Pour s'authentifier au lieu d'utiliser un secret
il utilise son certificat X509

Configuration Keycloak

- Déclarer un client avec Client authentication et Service Account Roles
- Dans l'onglet credentials, définir X509 et indiquer le Subject DN complet du certificat



Client credentials avec JWT-signé

Permet à un client de s'authentifier en utilisant un JWT signé avec une clé privée

Génération du JWT

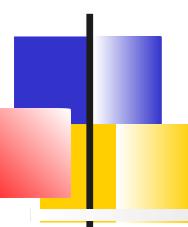
- Le client génère un JWT contenant des revendications (claims) spécifiques, généralement `client_id`, `iss`, `sub`, `aud` et `iat`.
- Le JWT est signé avec une clé privée détenue par le client. Le serveur vérifie l'intégrité et l'authenticité avec la clé publique correspondante.

Envoi du JWT :

- Le client envoie le JWT signé dans le cadre d'une requête d'authentification en utilisant généralement le champ `client_assertion` dans une requête de token.

Validation du JWT :

- Le serveur valide le JWT en vérifiant la signature avec la clé publique du client.
- Il vérifie également les revendications pour s'assurer que le JWT est valide et n'a pas expiré.
- Si le JWT est valide, le serveur d'autorisation procède à l'authentification du client et peut émettre un token d'accès.



Mise en œuvre dans Keycloak

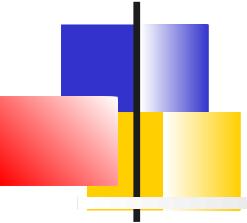
Dans l'onglet *Credentials*, sélectionner **Signed JWT** ou *Signed JWT with secret*

Dans l'onglet *Keys*, on peut générer ou importer une paire clé privée/clé publique.

Dans l'application cliente, générer un JWT et le signer avec la clé privée puis :

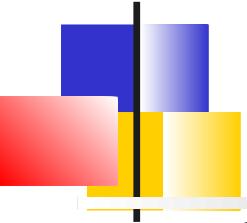
```
POST /realms/{realm-name}/protocol/openid-connect/token HTTP/1.1
Host: keycloak.example.com
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=client_credentials
&client_id={client-id}
&client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer
&client_assertion={signed-jwt}
```



Intégration Keycloak

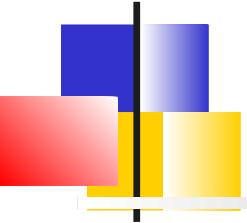
Introduction
Adaptateurs Keycloak
SpringBoot
Quarkus
Reverse Proxy



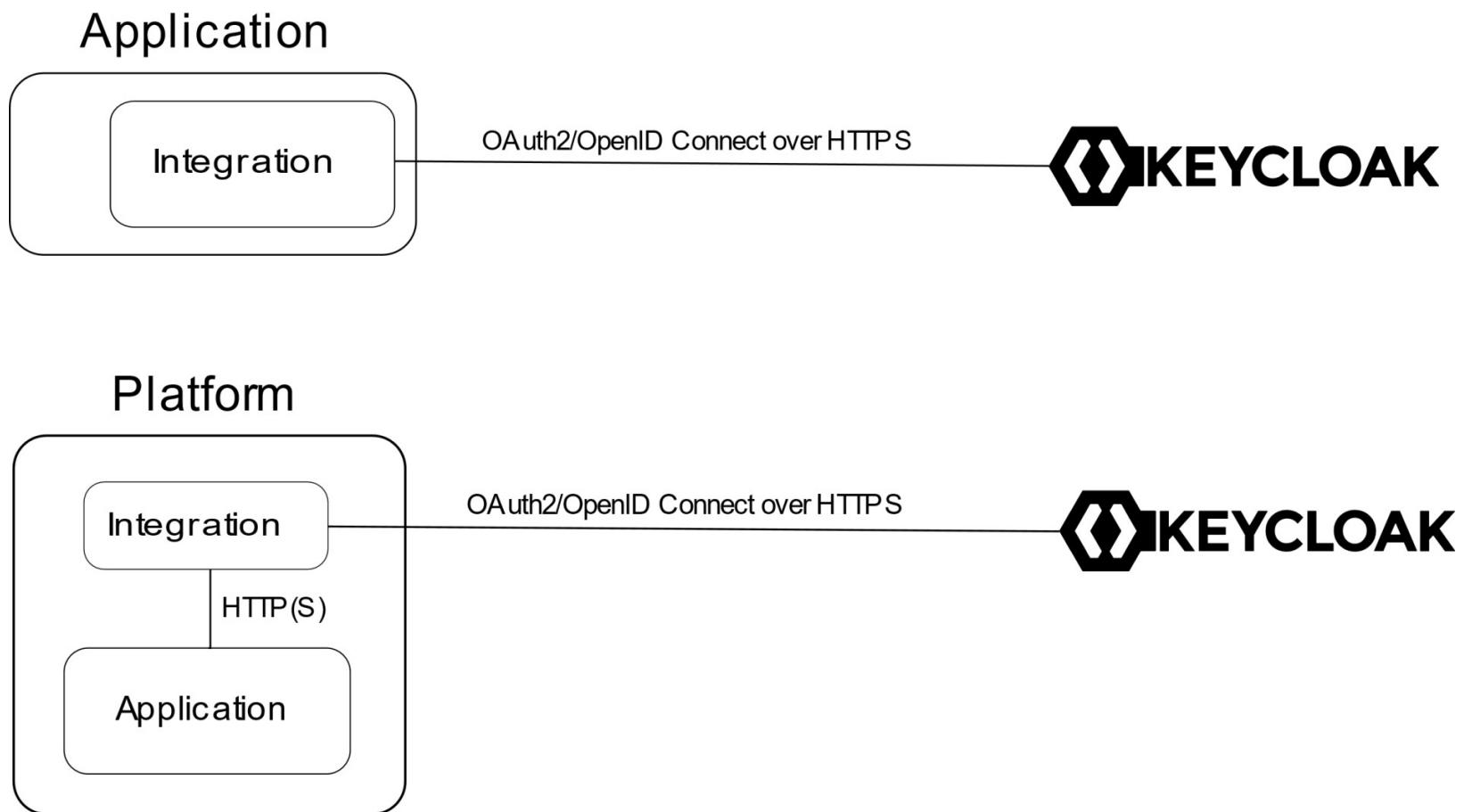
Introduction

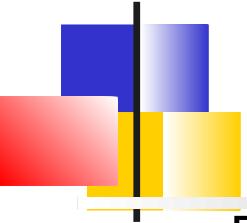
2 principaux styles d'intégration, selon l'emplacement du code d'intégration et de la configuration

- Embarqué : Le code embarque des librairies du langage ou du framework choisi
- Proxy : Utiliser un reverse proxy devant son application.
Code legacy, Gateway micro-service



Embarqué ou Proxy



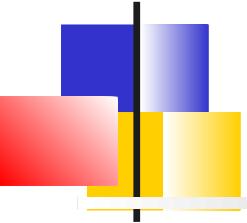


Librairies OpenID/oAuth

De nombreuses librairies bas niveau OpenID sont disponibles :

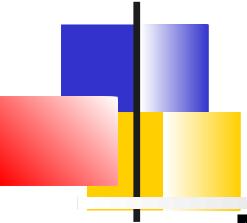
- C : *mod_auth_openidc*
- C# : *IdentityModel.OidcClient*
- Java : *GKIDP Broker*
- Golang : *OpenID Connect SDK*
- Javascript : *node openid-client*, *oidc-client-js*, *oauth4webapi*
- PHP : *phpOIDC 2016 Winter*
- Python : *OidcRP*, *pyoidc*
- Typescript : *angular-auth-oidc-client*

Elles sont référencées et certifiées par OpenIdConnect :
<https://openid.net/developers/certified/>



Intégration Keycloak

Introduction
Adaptateurs Keycloak
SpringBoot
Quarkus
Reverse Proxy

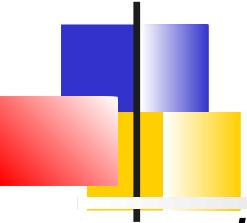


Introduction

Même si il sont toujours documentés, les adaptateurs Keycloak sont en cours de dépréciation.

Ils ont vocation à être remplacés par les frameworks de développement qui progressivement offre du support pour OIDC/oAuth2 donc Keycloak.

Seul l'adaptateur Javascript sera maintenu.

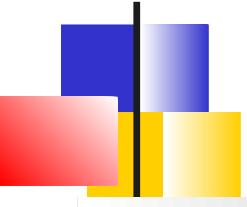


Adaptateurs Keycloak

Keycloak distribue des adaptateurs pour différentes plates-formes :

- Java : *JBoss EAP, WildFly, Tomcat, Jetty, Filtres Servlet, Spring Boot, Spring Security* (dépréciés)
- Javascript : Client-side
- Node.js (déprécié) : Server side
- Apache HTTP Server : *mod_auth_openidc*

Ces adaptateurs sont configurables via un simple fichier **keycloak.json** qui peut être généré via la console d'admin



Exemple adaptateur Javascript

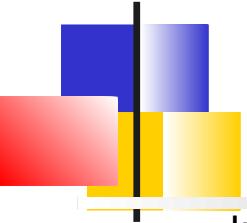
- 1) Charger la librairie via l'URL :

KC_URL/js/keycloak.js Ou npm install keycloak-js

- 2) Créer un objet *Keycloak* avec les informations du client et l'initialiser lorsque la fenêtre est chargée.

```
<head><script>
    function initKeycloak() {
        const keycloak = new Keycloak(); // Configuration avec keycloak.json
        keycloak.init({
            onLoad: 'login-required' // Nécessite le login lors du chargement de page
        }).then(function(authenticated) {
            alert(authenticated ? 'authenticated' : 'not authenticated');
        }).catch(function() {
            alert('failed to initialize');
        });
    }
</script></head><body onload="initKeycloak()">
```

- 3) Après l'authentification, le token est présent dans **keycloak.token**
req.setRequestHeader('Authorization', 'Bearer ' + keycloak.token);



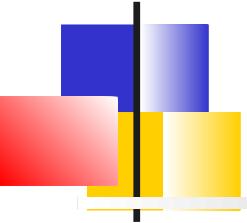
Configuration de l'objet keycloak

Instanciation : **url**, **realm** et **clientId**

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

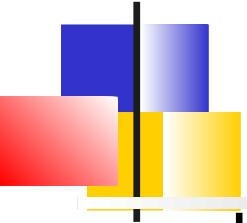
Paramètres de la fonction init() :

- **login-required** : Affiche la page de login si l'utilisateur n'est pas déjà authentifié
- **check-sso** : Succès seulement si l'utilisateur est déjà authentifié
- **checkLoginIframe** : Autorise une iframe qui détecte les logout sso
- **flow** : Le flow utilisé. Supporte également *implicit* et *hybrid*
- **pkceMethod** : Si on veut activer PKCE, seule valeur possible S256
- **enableLogging** : true pour activer les traces
- **scope** : Scopes demandés
- ...



Intégration Keycloak

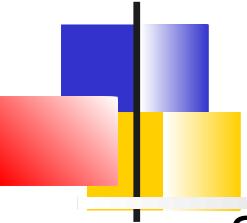
Introduction
Adaptateurs et Keycloak
SpringBoot
Quarkus
Reverse Proxy



Apports de SpringBoot

Le support de oAuth2 via Spring comprend 3 starters :

- **OAuth2 Client** : Intégration pour s'authentifier avec OpenIDConnect avec Google, Github, Facebook, Keycloak ...
- **OAuth2 Resource server** : Extraction du jeton et vérification des ACLs par rapport aux scopes client et/ou aux rôles contenu dans le jeton d'accès
- **OAuth2 Authorization server** : Un keycloak en Spring Boot (réapparu depuis peu)
- **Okta** : Pour travailler avec le fournisseur oAuth Okta (Pas disponible dans toutes les versions)



OpenIDConnect avec SpringBoot

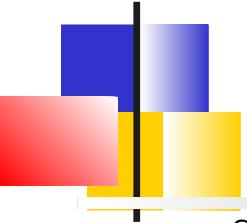
Starter ***oauht2-client***

@Bean

```
public SecurityWebFilterChain
securityWebFilterChain(ServerHttpSecurity http) {

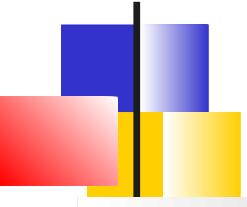
    return http.authorizeExchange()
        .anyExchange().authenticated()
        .and().oauth2Login()
        .and().csrf().disable()
        .build();

}
```



Configuration Keycloak

```
spring:
  security:
    oauth2:
      client:
        provider:
          keycloak:      # Ou tout simplement issuer-uri
            token-uri:
              http://localhost:8089/realms/<realm-name>/protocol/openid-
              connect/token
              authorization-uri: http://localhost:8089/realms/<realm-
              name>/protocol/openid-connect/auth
              user-info-uri: http://localhost:8089/auth/realms/<realm-
              name>/protocol/openid-connect/userinfo
              user-name-attribute: preferred_username
        registration: # Configuration client
          spring-app:
            provider: keycloak
            client-id: spring-app
            client-secret: 57abb4f6-5130-4c73-9545-6d377dd947cf
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/keycloak"
            scope :openid
```

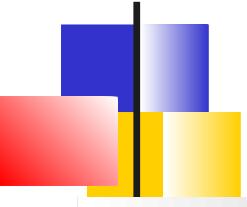


Accès à l'utilisateur loggé

```
@GetMapping("/oidc-principal")
public OidcUser getOidcUserPrincipal(
    @AuthenticationPrincipal OidcUser principal) {
    return principal;
}

...
Authentication authentication =
    SecurityContextHolder.getContext().getAuthentication();
if (authentication.getPrincipal() instanceof OidcUser) {
    OidcUser principal = ((OidcUser)
        authentication.getPrincipal()));

    // ...
}
```

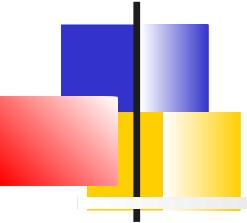


Logout

L'application Spring lors d'un logout doit se déconnecter sur Keycloak en utilisant le endpoint **/protocol/openid-connect/logout** et en fournissant dans le paramètre **id_token_hint** le jeton.

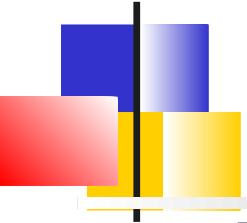
Cela est configuré via un **LogoutHandler** de Spring :

```
httpSecurity.logout(logout -> logout.logoutUrl("/logout") .  
    addLogoutHandler(keycloakLogoutHandler) .  
    invalidateHttpSession(true) .  
    logoutSuccessUrl("/"))
```



LogoutHandler

```
private void logoutFromKeycloak(OidcUser user) {  
    String endSessionEndpoint = user.getIssuer() + "/protocol/openid-connect/logout";  
  
    UriComponentsBuilder builder = UriComponentsBuilder  
        .fromUriString(endSessionEndpoint)  
        .queryParam("id_token_hint", user.getIdToken().getTokenValue());  
  
    ResponseEntity<String> logoutResponse = restTemplate.getForEntity(  
        builder.toUriString(), String.class;  
    logger.info("Redirecting to " + builder.toUriString());  
    if (logoutResponse.getStatusCode().is2xxSuccessful()) {  
        logger.info("Successfully logged out from Keycloak");  
    } else {  
        logger.error("Could not propagate logout to Keycloak");  
    }  
}
```

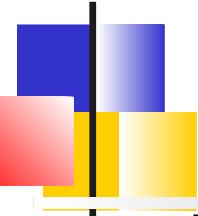


SSO-logout

Pour gérer le sso (logout initié par un autre application), il faut utiliser un backchannel logout

```
@PostMapping("/sso-logout")
public void ssoLogout(@RequestParam("logout_token")
    String logoutToken, HttpSession session) {

    session.invalidate();
}
```

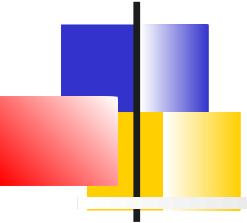


Serveur de ressources

Dépendance : **oauth2-resource-server**

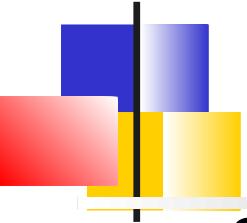
Le serveur de ressources doit vérifier la signature du jeton pour s'assurer que les données n'ont pas été modifiées.

- **jwk-set-uri** contient la clé publique que le serveur peut utiliser pour la vérification
- **issuer-uri** pointe vers l'URI de Keycloak. Utilisé pour la découverte de *jwk-set-uri*



Exemple *application.yml*

```
spring:  
  security:  
    oauth2:  
      resourceserver:  
        jwt:  
          # issuer-uri: http://keycloak:8083/realmns/<realm-name>  
          jwk-set-uri: http://keycloak:8083/realmns/<realm-name>/protocol/openid-connect/certs
```



Scope vs Spring Authority

Spring ajoute des autorités¹ au principal en fonction des scopes présents dans le jeton

Les autorités correspondant aux scopes sont préfixées par "SCOPE_".

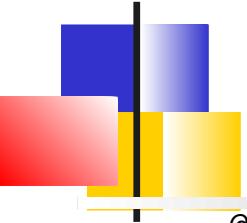
Par exemple, la portée *openid* devient une autorité accordée SCOPE_openid.

Il ajoute également l'autorité *OIDC_USER*

Cela peut être adapté via un bean

JwtAuthenticationConverter

- Positionner les Authorities à partir d'autre(s) Claim
- Changer le préfixe utilisés



Exemple pour Keycloak

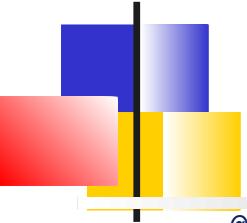
```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverterForKeycloak() {

    Converter<Jwt, Collection<GrantedAuthority>> jwtGrantedAuthoritiesConverter = jwt -> {
        Map<String, Collection<String>> realmAccess = jwt.getClaim("realm_access");
        Collection<String> roles = realmAccess.get("roles");
        return roles.stream()
            .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
            .collect(Collectors.toList());
    };

    var jwtAuthenticationConverter = new JwtAuthenticationConverter();

    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(jwtGrantedAuthoritiesConver-
ter);

    return jwtAuthenticationConverter;
}
```

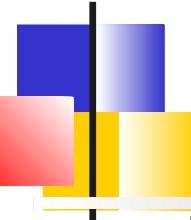


Configuration typique

SpringBoot

@Configuration

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http.cors()  
            .and()  
            .authorizeRequests()  
                .antMatchers(HttpMethod.GET, "/user/info", "/api/foos/**")  
                    .hasAuthority("SCOPE_read")  
                .antMatchers(HttpMethod.POST, "/api/foos")  
                    .hasAuthority("SCOPE_write")  
                .anyRequest()  
                    .authenticated()  
            .and()  
            .oauth2ResourceServer()  
                .jwt();  
    }  
}
```



Adaptateur SpringBoot

Keycloak a un adaptateur pour SpringBoot mais il est déprécié

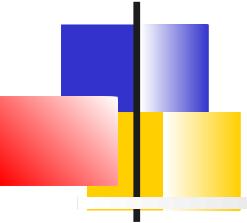
org.keycloak:keycloak-spring-boot-starter

Support pour *Tomcat, Undertow, Jetty*

La configuration s'effectue via le fichier de configuration
SpringBoot

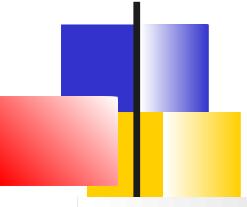
Les ACLS peuvent facilement s'appuyer sur les rôles

```
keycloak.realm = demorealm
keycloak.auth-server-url = http://127.0.0.1:8080
keycloak.ssl-required = external
keycloak.resource = demoapp
keycloak.credentials.secret = 11111111-1111-1111-1111-111111111111
keycloak.use-resource-role-mappings = true
keycloak.securityConstraints[1].authRoles[0] = admin
keycloak.securityConstraints[1].securityCollections[0].name = admin stuff
keycloak.securityConstraints[1].securityCollections[0].patterns[0] = /admin
```



Intégration Keycloak

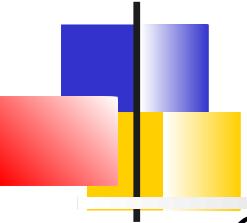
Introduction
Adaptateurs Keycloak
SpringBoot
Quarkus
Reverse Proxy



Extensions oidc

Différentes extensions Quarkus peuvent être utilisées en fonction des usages :

- **quarkus-oidc** : fournit un adaptateur OpenID supportant l'extraction du jeton et l' Authorization Code Flow
- **quarkus-oidc-client** : Permet d'obtenir des tokens d'accès et de rafraîchissement auprès de fournisseur supportant les grant type : *client-credentials*, *password* et *refresh_token*
- **quarkus-oidc-token-propagation** : Dépend de *quarkus-oidc*
Permet de propager un jeton d'accès aux services en aval
- **quarkus-oidc-client-filter** : Dépend de *quarkus-oidc-client*
Positionne le jeton obtenu par *oidc-client* dans l'entête d'Authorization pour faire des appels REST vers des services aval



DevServices et Keycloak

Quarkus propose une auto-configuration pour Keycloak,
=> Pas besoin d'installer Keycloak sur son poste

Activé si

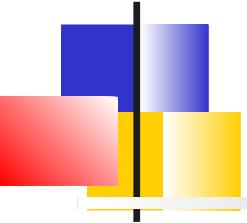
- *quarkus-oidc*
- dev et test mode
- La propriété *quarkus.oidc.auth-server-url* n'est pas configuré

Le service démarre un container Keycloak et l'initialise en
important ou créant un realm.

Le realm peut-être spécifié par :

`quarkus.keycloak.devservices.realm-path=quarkus-realm.json`

De plus, la Dev UI (`/q/dev`) permet d'acquérir les jetons de
Keycloak et de tester l'application Quarkus.

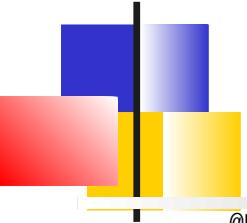


microprofile-jwt

Quarkus demande le profil
microprofile-jwt

Ce profil génère les rôles utilisateur de Keycloak dans l'attribut **groups** du jeton.

Quarkus mappe l'attribut groups à sa propre notion de rôle



Exemple Service REST

```
@Path("/api/users")
public class UsersResource {

    @Inject
    SecurityIdentity securityIdentity;

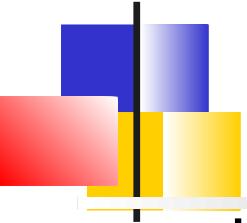
    @GET
    @Path("/me")
    @RolesAllowed("user")
    @NoCache
    public User me() {
        return new User(securityIdentity);
    }

    public static class User {

        private final String userName;

        User(SecurityIdentity securityIdentity) {
            this.userName = securityIdentity.getPrincipal().getName();
        }

        public String getUserName() {
            return userName;
        }
    }
}
```

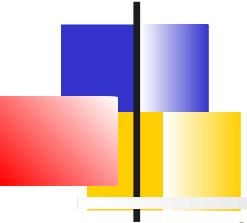


Client REST utilisant les jetons oAuth2

Un client REST voulant accéder à une ressource protégée par oAuth2 doit utiliser un jeton.

2 scénarios

- Un client obtient son propre jeton et l'utilise dans ses appels rest :
oidc-client +
oidc-client-reactive-filter ou oidc-client-filter
un micro-service accédant à un autre micro-service
- Le token courant (obtenu par oidc) est propagé
oidc-token-propagation ou oidc-token-propagation-reactive
Cas de la gateway par exemple



Configuration oidc-client

Découverte automatique

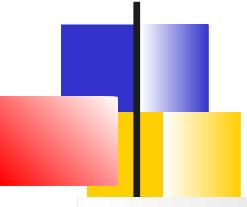
```
quarkus.oidc-client.auth-server-url=  
  http://localhost:8180/auth/realms/quarkus
```

Découverte manuelle

```
quarkus.oidc-client.discovery-enabled=false  
quarkus.oidc-client.token-path=  
  http://localhost:8180/auth/realms/quarkus/protocol/openid-c  
onnect/tokens
```

grant type client_credentials

```
quarkus.oidc-client.auth-server-url=http://localhost:8180/realms/quarkus/  
quarkus.oidc-client.client-id=quarkus-app  
quarkus.oidc-client.credentials.secret=secret
```



Utilisation dans le RestClient

Extensions : ***quarkus-oidc-client-reactive-filter*** ou
quarkus-oidc-client-filter

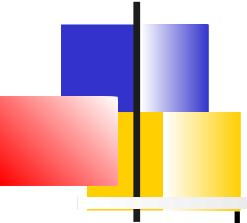
Les extensions fournissent

OidcClientRequestReactiveFilter ou

OidcClientRequestFilter qui peuvent être appliqués
à une interface RestClient

```
@RegisterRestClient
@RegisterProvider(OidcClientRequestReactiveFilter.class)
@Path("/")
public interface ProtectedResourceService {

    @GET
    Uni<String> getUserName();
}
```



Propagation de jeton

L'extension **quarkus-oidc-token-propagation** fournit 2 implémentations de *ClientRequestFilter* qui simplifient la propagation des informations d'authentification

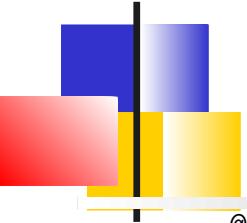
- **AccessTokenRequestFilter** propage le jeton Bearer présent dans la requête en cours
- **JsonWebTokenRequestFilter** fournit la même fonctionnalité, mais fournit en plus la prise en charge des jetons JWT

Cette extension est typiquement utilisée :

- Pour propager le jeton venant d'être obtenu par l'Authorization Code Flow.
Ex : requête initiale Gateway
- Pour propager le jeton présent dans le Bearer
le même jeton circule dans toute l'architecture micro-services



Enregistrement du filtre AccessToken



```
@RegisterRestClient
@AccessToken
@Path("/")
public interface ProtectedResourceService {
```

```
    @GET
    String getUserName();
}
```

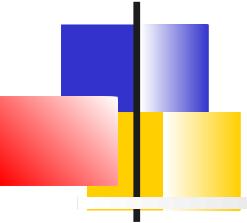
Ou

```
@RegisterRestClient
@RegisterProvider(AccessTokenRequestFilter.class)
@Path("/")
public interface ProtectedResourceService {
```

```
    @GET
    String getUserName();
}
```

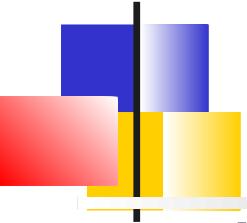
Ou automatiquement si configuration suivante :

```
quarkus.oidc-token-propagation.register-filter=true
quarkus.oidc-token-propagation.json-web-token=false
```



Intégration Keycloak

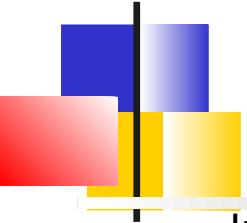
Introduction
Librairies et adaptateurs
SpringBoot
Quarkus
Reverse Proxy



Introduction

La prise en charge d'OpenID Connect et d'OAuth2 est une fonctionnalité obligatoire pour les proxys.

- Apache HTTP Server et Nginx, fournissent les extensions nécessaires pour ces protocoles

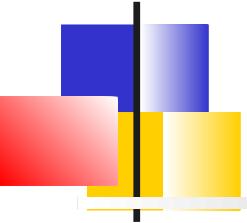


Exemple Apache

Installation du module ***mod_auth_oidc***¹

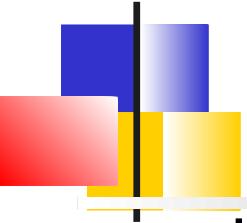
Puis configuration

```
LoadModule auth_openidc_module modules/mod_auth_openidc.so
ServerName localhost
<VirtualHost *:80>
    ProxyPass / http://localhost:8000/
    ProxyPassReverse / http://localhost:8000/
    OIDCCryptoPassphrase CHANGE_ME
    OIDCProviderMetadataURL http://keycloak/realm/<realm-name>/.well-known/openid-configuration
    OIDCClientID mywebapp
    OIDCClientSecret CLIENT_SECRET
    OIDCRedirectURI http://localhost/callback
    OIDCCookieDomain localhost
    OIDCCookiePath /
    OIDCCookieSameSite On
    <Location />
        AuthType openid-connect
        Require valid-user
    </Location>
</VirtualHost>
```



Stratégies d'autorisation

RBAC, GBAC, OAuth2 scopes
Authorization service



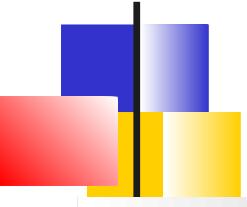
Introduction

Les données relatives à l'autorisation (l'utilisateur peut-il accéder à une ressource) sont extraites du jeton fourni par Keycloak : les revendications

- Cela peut être n'importe quel revendication
- En général, on utilise les rôles, les groupes ou les scopes

2 stratégies pour implémenter l'autorisation :

- Implémenter le contrôle d'accès au niveau de l'application déclarativement ou programmatiquement
- Déléguer les décisions d'accès à un service externe



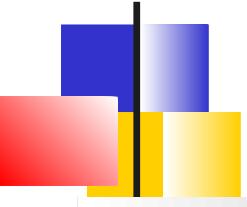
RBAC

RBAC (Role Base Access Control) vous permet de protéger les ressources en fonction de l'attribution ou non d'un rôle à l'utilisateur.

Keycloak a un support pour la gestion des rôles, ainsi que pour propager ces rôles dans les jetons

Les rôles représentent généralement un rôle qu'un utilisateur a

- dans l'organisation : Realm roles dans Keycloak
- ou dans le contexte d'une application :

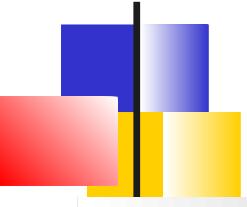


Gestion des rôles

Pour éviter l'explosion de rôles, n'utilisez pas les rôles pour une autorisation fine.

Keycloak propose 2 mécanismes pour faciliter la gestion des rôles :

- Un rôle peut être affecté à un groupe.
=> Tous les membres du groupe ont le rôle
- Un rôle peut être composite. Il englobe plusieurs rôles.



Groupes

Une stratégie **GBAC (Group Base Access Control)** définit des ACLs par rapport aux groupes

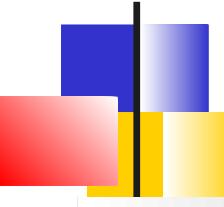
Les groupes représentent généralement le service auquel appartient un utilisateur

Les groupes sont hiérarchiques, ce sont une cartographie de l'organigramme d'une organisation

Les groupes ne sont pas automatiquement inclus dans les jetons. Cela nécessite un mapping particulier

Dans les jetons, les groupes sont représentés par un chemin

- Ex : */human resource/manager*



Modèle oAuth2

Dans le modèle oAuth2, les ACLs sont définis en fonction des scopes
=> l'autorisation est uniquement basée sur le consentement de l'utilisateur.

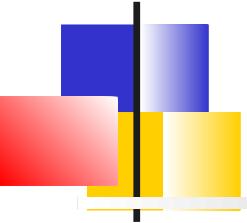
=> Les ACLs protège alors en fonction du client plutôt que de l'utilisateur

L'objectif principal étant de protéger les informations des utilisateurs plutôt que les ressources du serveur de ressources.

Par défaut, les clients de Keycloak n'utilisent pas le consentement de l'utilisateur.

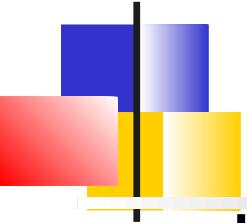
Les services (APIs) peuvent donc naturellement configurer des ACLs en fonction du client qui les accède :

- Client interne / externe
- Outil de monitoring / service applicatif



Stratégies d'autorisation

RBAC, GBAC, OAuth2 scopes
Authorization service

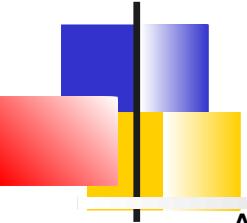


Introduction

L'autorisation centralisée permet d'externaliser la gestion des accès à l'aide d'un service d'autorisation externe.

Elle permet d'utiliser plusieurs mécanismes de contrôle d'accès sans y coupler l'application.

Keycloak peut agir comme un service d'autorisation centralisé via une fonctionnalité appelée **Authorization Services**.



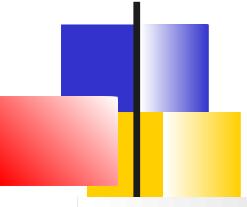
Introduction

Approche RBAC classique :

```
If (User.hasRole("manager")) {  
    // can access the protected resource  
}  
=> Si l'on veut changer les règles d'accès (ajout d'un nouveau rôle par exemple), il faut  
    modifier le code et redéployer
```

Approche Autorisation centralisée

```
If (User.canAccess("Manager Resource")) {  
    // can access the protected resource  
}  
=> aucune référence à un mécanisme de contrôle d'accès spécifique ; le contrôle d'accès est  
    basé sur la ressource.  
=> Les modifications apportées à l'accès à Manager Resource se définissent dans Keycloak et  
    n'ont pas d'impact sur le code applicatif
```

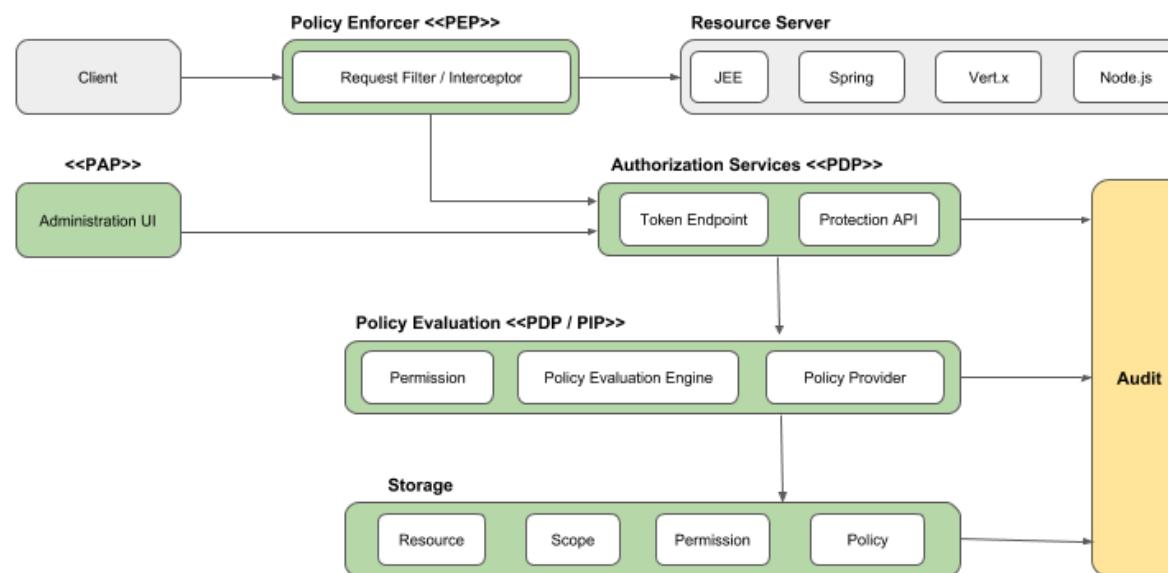


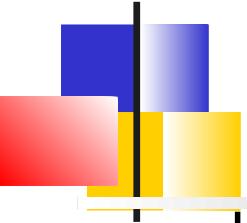
Fonctionnalités

Keycloak authorization services est capable de combiner différent mécanisme de contrôle d'accès :

- ABAC
- RBAC
- UBAC (User-based access control)
- CBAC (Context-based access control)
- Rule-based access control
- Utilisation de JavaScript pour évaluer les décisions
- Time-based access control
- Custom Access Control via un SPI

Architecture

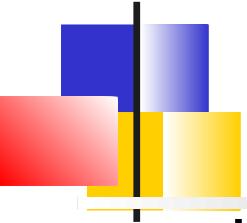




Configuration

La configuration s'effectue en 3 étapes :

- 1) **Définition des ressources** via la console d'admin ou l'API:
Resource Server <=> Client
Ressources <=> Motif d'URI
Scopes (Optionnel) <=> Action sur la ressource
- 2) **Définition des policy et des permissions** via la console d'admin ou l'API
policy : conditions devant être satisfaites
permissions : Associe une Ressource et éventuellement un scope à un ensemble de *policies*
- 3) **Appliquer les ACLs** sur le serveur de ressources. Ceci est réalisé en activant un point d'application (PEP) sur le serveur de ressources qui est capable de communiquer avec le serveur d'autorisation.
Keycloak fournit des implémentations pour ses clients Adapter



Policy Enforcer Point

Un PEP est un filtre ou intercepteur dans l'application qui vérifie les ACLs.

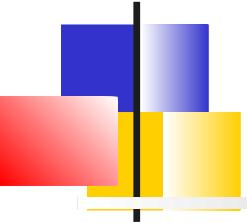
Les autorisations sont appliquées en fonction du protocole que l'on utilise :

- Si UMA¹, le PEP attend un jeton particulier contenant les permissions résolues : le RPT²
Le RPT s'obtient avec un jeton d'accès standard
- Sinon, le PEP envoie le jeton d'accès standard au serveur d'autorisation pour résoudre les ACLs

1. <https://docs.kantarainitiative.org/uma/wg/rec-oauth-uma-grant-2.0.html>

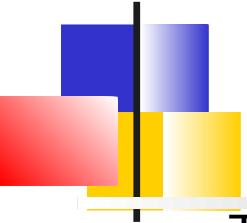
2. Requesting Party Token





Maximiser la sécurité des standards

Modèle de menaces - RFC 9700
Preuve de possession
FIPS
FAPI



Contexte de RFC 9700

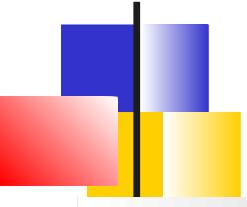
Titre : "OAuth 2.0 Security Best Current Practice"

Publiée en janvier 2025

Actualise les recommandations de sécurité pour OAuth 2.0, en tenant compte des retours d'expérience accumulés depuis la publication initiale d'OAuth

Le RFC 9700 :

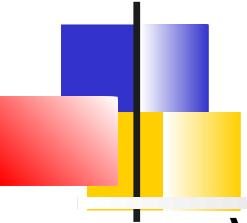
- Actualise les modèles de menaces (au-delà des RFC 6749, 6750, 6819),
- Propose de nouvelles contre-mesures face aux attaques connues et émergentes,
- Déprécie certains flux jugés peu sûrs (notamment le Implicit Grant),
- Introduit des exigences de sécurité renforcées, en anticipation de OAuth 2.1.



Menaces

Le RFC identifie 5 types d'attaquants (A1-A5), allant de l'attaquant web basique (exécutant un client malveillant) au network attacker capable d'intercepter ou modifier des communications. Il inclut :

- Les attaques via Referer ou historique du navigateur
Contre mesures : CSP, Referrer-Policy: strict-origin, éviter le stockage des token dans l'url
- Les attaques par mix-up, le code/jeton est renvoyé à un serveur d'autorisation malveillant
Contre mesure : Rediect URI, claim iss
- L'injection de code d'autorisation ou de tokens d'accès,
- Les attaques sur les flux de communication via navigateur (postMessage...), Exploitation des communications inter-origines dans le navigateur.
- Le PKCE downgrade attack, où l'attaquant tente de forcer une négociation sans PKCE.



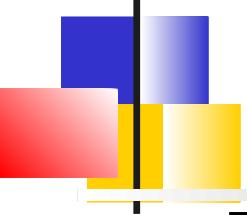
Protection des flux basés sur la redirection

Validation stricte des URI de redirection : utilisation de correspondance exacte, pas de motifs avec jokers, sauf pour les ports dans les apps natives.

Éviter les redirections ouvertes : clients et serveurs doivent empêcher les redirections vers des URI non vérifiés.

Protection contre les attaques CSRF :

- Via state + token CSRF à usage unique,
- Ou via PKCE (Proof Key for Code Exchange),
- Ou via nonce en OpenID Connect (Implémenté nativement dans Keycloak)

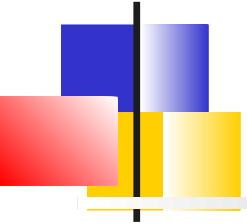


Injection et Rejet de Code d'autorisation

Tous les clients (même confidentiels) doivent implémenter PKCE.

L'usage de méthodes de challenge sûres (S256) est obligatoire.

Les serveurs doivent refuser toute tentative d'échange de code si *code_challenge* est attendu mais absent.



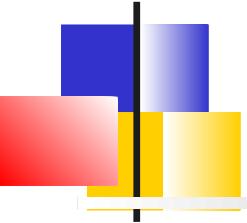
Prévention du rejet des tokens

Access Tokens : contraindre leur usage à l'émetteur via une preuve de possession :

- mutual TLS (RFC8705),
- ou DPoP (RFC9449).

Refresh Tokens :

- Doivent être tournants (rotation) ou contraints à un émetteur unique.
- Interdiction d'utiliser les mêmes tokens entre clients.



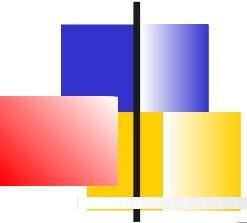
Restrictions des privilèges des tokens

Restreindre les tokens à un périmètre minimum
(scopes)

Imposer une audience (aud) claire et vérifiée côté
ressource

Utiliser *authorization_details*¹ (RFC9396) si
nécessaire.

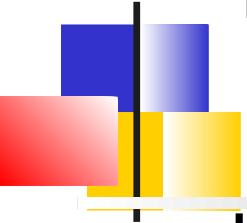
1. *Claim utilisé pour spécifier des détails sur les ressources auxquelles le client souhaite accéder, les actions qu'il souhaite effectuer, et d'autres métadonnées pertinentes.*



Authentification des clients

Fortement recommandé d'utiliser une authentification asymétrique :

- TLS mutuel, ou
- JWT signé (`private_key_jwt`).



Métadonnées et configuration automatique

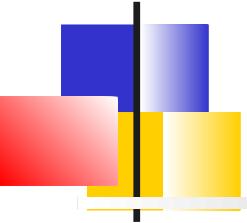
Utiliser Authorization Server Metadata (RFC8414).

Accès des méta-données de configuration du serveur via :

`/ .well-known/openid-configuration`
et
`/ .well-known/oauth-authorization-server`

Avantages :

- Réduction des erreurs de configuration,
- Détection automatique des capacités de sécurité (PKCE, algorithmes supportés...).



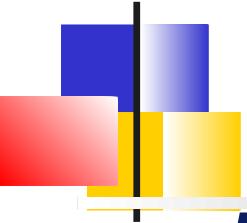
Maximiser la sécurité des standards

Modèle de menaces – RFC 9700

Preuve de possession

FIPS

FAPI



Introduction - DPoP

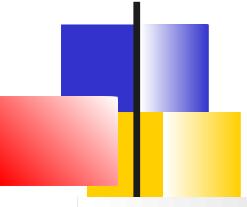
DPoP - Demonstration of Proof of Possession est définie via la RFC 9449

C'est un mécanisme léger pour démontrer que le client possède une clé privée liée au token

Recommandé par la RFC 9700

Alternative légère à mTLS, adaptée aux apps mobiles, SPA, etc.

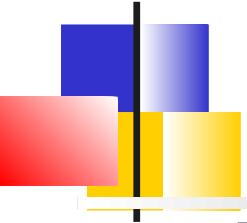
Objectif : éviter le rejet (replay) d'un jeton d'accès intercepté par un attaquant. Le jeton devient inutile sans la clé privée du client.



Mécanismes

- 1) Le client génère une paire de clés asymétriques (EC ou RSA)
- 2) Lors de la requête d'authentification :
Il envoie sa clé publique encodée dans un header DPoP (JWT signé)
Le serveur d'autorisation enregistre le thumbprint de cette clé et
l'insère dans l'access token sous le claim ***cnf.jkt***
- 3) Lors de chaque appel API avec le jeton d'accès :
Le client signe un nouveau JWT DpoP qui contient la clé publique
Ce JWT est ajouté dans l'en-tête de la requête
- 4) Le serveur de ressources :
Vérifie que le JWT DPoP est valide
Vérifie que le token d'accès a été émis pour cette clé en recalculant le
thumbprint avec le même algorithme

Résultat : l'appel ne peut être effectué que par le détenteur de la clé privée.



DpoP et Keycloak

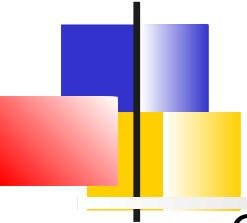
Depuis la v22+, Keycloak supporte DpoP en mode Preview

Ce mécanisme est désactivé par défaut.

Pour l'activer :

```
kc.sh start --features=dpop
```

Ensuite pour un client, activer DpoP dans la configuration



Implémentation côté client

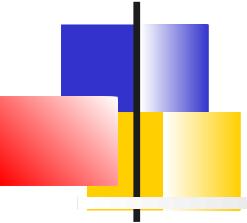
Générer une clé privée/publique (ex : EC P-256)

- Construire un JWT DPoP signé avec la clé privée
- Inclure les claims : htu, htm, jti, iat
- Ajouter l'en-tête DPoP à chaque requête vers l'API
- Stocker et réutiliser la clé

Librairies utiles :

- JavaScript jsonwebtoken (npm)
- Java jjwt (Java JWT), Nimbus JOSE+JWT
- Python PyJWT
- C#/.NET System.IdentityModel.Tokens.Jwt

L'en-tête DPoP doit être généré pour chaque appel, car le JWT est horodaté (iat) et lié à l'URL (htu).



Exemple avec jjwt

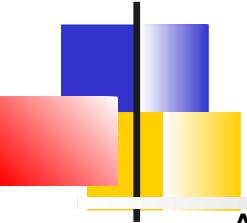
```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize(2048);
KeyPair keyPair = keyGen.generateKeyPair();

// Construction du JWK (simplifié pour l'exemple)
Map<String, Object> jwk = new HashMap<>();
jwk.put("kty", "RSA");
jwk.put("n", Base64.getUrlEncoder().withoutPadding() .encodeToString(keyPair.getPublic().getEncoded()));
jwk.put("e", "AQAB");

// Header personnalisé DPoP
Map<String, Object> header = new HashMap<>();
header.put("typ", "dpop+jwt");
header.put("alg", "RS256");
header.put("jwk", jwk);

// Claims DPoP requis
Map<String, Object> claims = new HashMap<>();
claims.put("jti", UUID.randomUUID().toString());
claims.put("htm", "POST");
claims.put("htu", "https://exemple.com/oauth2/token");
claims.put("iat", System.currentTimeMillis() / 1000);

// Génération du JWT signé
String jwt = Jwts.builder()
    .setHeader(header)
    .setClaims(claims)
    .signWith(keyPair.getPrivate(), SignatureAlgorithm.RS256)
    .compact();
```



Avantages et limites

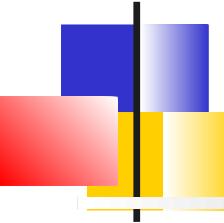
Avantages :

- Légèreté → pas besoin de certificat client ou infrastructure PKI
- Facilement intégrable dans des applications SPA ou mobiles
- Réduit considérablement le risque de rejet de jeton

Limites :

- Ne sécurise que les clients honnêtes (pas d'attestation du matériel comme mTLS)
- Nécessite une gestion locale de clés (durée de vie, persistance)
- Dépend du support correct côté ressource server

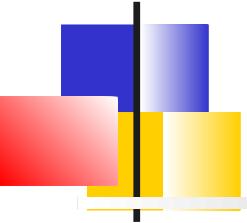
Pour les environnements critiques (FAPI Advanced), DPoP peut être insuffisant → préférer mTLS.



Preuve de possession avec mTLS

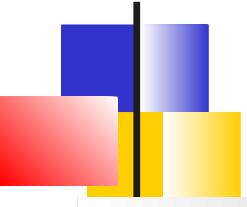
Il est également possible d'effectuer la preuve de possession avec mTLS

- Il faut configurer Keycloak afin qu'il associe le certificat client au jeton.
Advanced – Advanced Settings - OAuth 2.0 Mutual TLS Certificate Bound Access Tokens Enabled
- Keycloak renvoie alors un jeton avec le claim cnt contenant l'empreinte du certificat (thumbprint)
- Lorsque le serveur d'api reçoit le jeton un peut vérifier le certificat client et le thumbprint



Maximiser la sécurité des standards

Modèle de menaces – RFC 9700
Preuve de possession
FIPS
FAPI



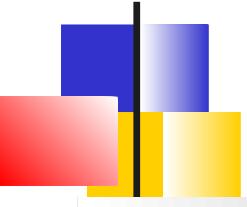
Introduction

FIPS (Federal Information Processing Standards)

est un ensemble de standards de sécurité informatique définis par le gouvernement des États-Unis.

Le plus connu est FIPS 140-2 / 140-3, qui spécifie les exigences pour les modules cryptographiques (logiciels ou matériels).

Objectif : Assurer que les algorithmes cryptographiques utilisés sont robustes, éprouvés et conformes



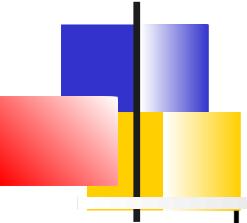
Implications

Interdiction des algorithmes faibles ou obsolètes (ex : MD5, SHA-1, RSA 1024)

Imposition de l'usage de :

- RSA \geq 2048 bits
- AES-256 / GCM
- SHA-256, SHA-384, SHA-512
- Signatures PSS (RSA-PSS)

Nécessité d'utiliser une JVM et des fournisseurs cryptographiques compatibles FIPS



Conditions pour keycloak

Le serveur doit tourner sur un système FIPS-activé (ex. RHEL/Fedora, ou Ubuntu FIPS via Ubuntu Pro)

Keycloak utilise un Java FIPS-compliant via OpenJDK.

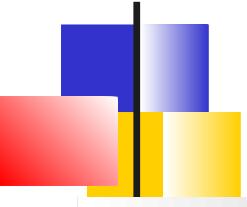
BouncyCastle FIPS : la version par défaut de BouncyCastle utilisé pour la cryptography n'est pas FIPS compliant.

=> Il faut installer les librairies adéquates dans
KEYCLOAK_HOME/providers :

- *bc-fips-2.0.0.jar*
- *bctls-fips-2.0.19.jar*
- *bcpkix-fips-2.0.7.jar*

Le keystore doit être compatible FIPS :

- PKCS12 : fonctionne en mode non-approved
- BCFKS : requis pour le mode FIPS strict



Démarrage

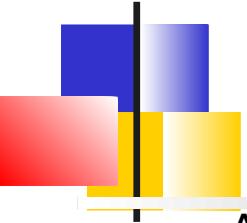
Une fois les conditions remplies, on peut démarrer keycloak en mode **fips** ou en **strict fips**.

Modifier le fichier *java.security* afin que la JVM ne tente pas un algorithme interdit. Ex :

```
security.provider.1=org.bouncycastle.jcajce.provider.BouncyCastleFipsProvider  
security.provider.2=org.bouncycastle.jsse.provider.BouncyCastleJsseProvider  
security.overridePropertiesFile=true  
  
export JAVA_OPTS_APPEND="-Djava.security.properties=/opt/keycloak/conf/fips-java.security"
```

Activer la features fips et éventuellement positionner le mode script

```
bin/kc.[sh|bat] start --features=fips --fips-mode=strict --hostname=localhost \  
--https-key-store-password=passwordpassword \  
--log-level=INFO,org.keycloak.common.crypto:TRACE,org.keycloak.crypto:TRACE
```



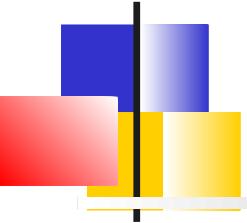
Impacts de FIPS sur Keycloak et les applications

Avantages

- Conformité réglementaire (USA, institutions financières ou gouvernementales)
- Usage exclusif d'algorithmes robustes
- Accroît la sécurité globale de votre chaîne d'authentification
- Souvent exigé pour FAPI Advanced, OpenBanking, ou projets certifiables

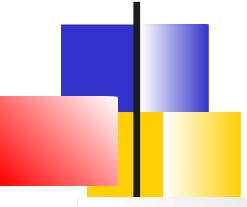
Limitations / Effets de bord

- Algorithmes : Algos faibles désactivés (ex. SHA-1, RSA-1024)
- Signatures : Certaines signatures interdites (ex. none)
- Tokens JWT : JWT doivent être signés avec algo FIPS-compatibles
- Clients existants : Incompatibles si ils utilisent un algo non autorisé
- Intégrations tierces Doivent être compatibles FIPS



Maximiser la sécurité des standards

Modèle de menaces – RFC 9700
Preuve de possession
FIPS
FAPI



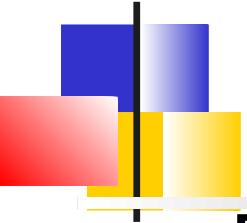
Introduction

FAPI (Financial-grade API) est un ensemble de spécifications de sécurité renforcées, conçues pour protéger les échanges OAuth2 / OIDC dans des contextes hautement sensibles, en particulier dans :

- la finance (Open Banking UK, Berlin Group, PSD2)
- la santé
- les données personnelles critiques

Apports :

- Des profils renforcés d'OAuth2 / OIDC
- Des exigences sur les clients, tokens, signatures, preuves de possession
- Une standardisation du niveau de sécurité pour des appels API critiques



Profils FAPI

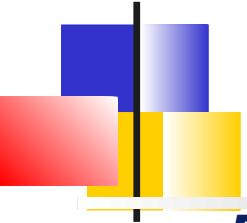
FAPI 2.0 combine plusieurs recommandations (RFC 9207, RFC 8705, RFC 8707, DPoP...) et impose un ensemble de contraintes, regroupées dans deux profils principaux :

- ***Baseline***

Renforce OAuth/OIDC standard
Clients web ou natifs sécurisés

- ***Advanced***

Sécurité maximale (certification)
Services financiers, eIDAS, etc.



Principales exigences FAPI

PKCE obligatoire : Même pour clients confidentiels

Preuve de possession (DPoP ou mTLS) : Obligatoire pour les tokens

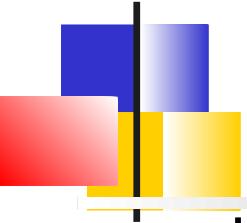
Signed Authorization Response (JARM) : Réponse d'autorisation encapsulée dans un jeton JWT signée (*response_mode=jwt* dans la requête initiale)

Claims précises (*aud, nonce, acr, client_id*) : Présence vérifiée et horodatée

Flows interdits : *implicit, password, client_credentials* (selon profil)

ID Token signé obligatoirement : Pas de token non signé ou avec alg: none

Refresh tokens bound (DpoP/mTLS) : Refresh token lié à une clé, même pour clients publics

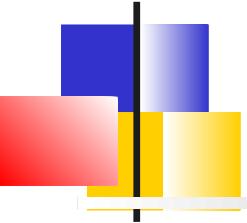


FAPI et Keycloak

Keycloak est conforme FAPI1 depuis la version 15.0.2, il n'est pas officiellement certifié FAPI-2.0

Via les ***client policies***, Keycloak permet d'imposer des exigences FAPI (Baseline, Advanced, etc.) sans avoir à tout configurer manuellement pour chaque client.

Ces politiques contrôlent dynamiquement les conditions d'enregistrement, d'authentification et de traitement des clients en fonction de règles (policies) et de profils (profiles).



Mise en œuvre

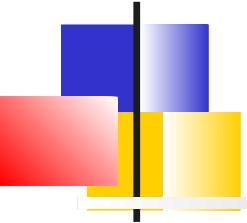
Admin → Realm Settings → Client Policies

Créer une politique et définir les clients qui y seront associés (tous, par rôle, par client_id, par client_scope)

Appliquer un profil client

- En utilisant les profils définis
- Ou un profil custom qui spécifie chaque feature requises (*pkce-enforcer*, *dpop-enforcer*, *mtls-holder-of-key-enforcer*, ...)

Tout client qui correspond à la policy définie sera soumis automatiquement aux règles FAPI imposées.



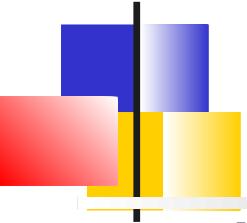
Administration Keycloak

Organisation

Gestion des utilisateurs

Flow d'authentification

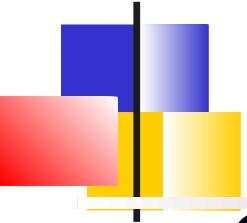
Gestion des sessions et jetons



Introduction

Une organisation relativement classique consiste à avoir :

- 1 ou plusieurs super-admin
- Des administrateurs dédiés à des realms
- Éventuellement, parmi les administrateurs dédiés des sous-rôles déterminant des permissions limitées

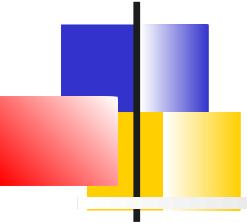


Attribution des droits d'administration

Chaque realm dispose de son propre client **realm-management**, qui contient des rôles d'administration.

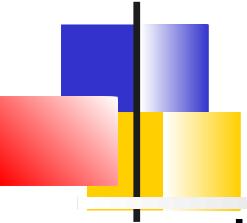
Un utilisateur devient admin d'un realm lorsqu'il reçoit un ou plusieurs rôles de ce client.

- **realm-admin** : Admin complet sur le realm
- **manage-users** : Gérer les utilisateurs
- **view-clients** : Lire les clients
- **manage-roles** : Gérer les rôles



Administration Keycloak

Organisation
Gestion des utilisateurs
Flow d'authentification
Gestion des sessions et jetons

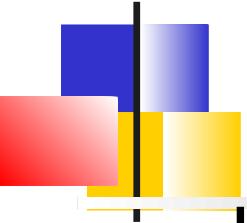


Utilisateurs locaux et externes

Keycloak gère une base utilisateur dans sa base de données. Les opérations de gestion concerne :

- La création d'utilisateur
- La gestion de leurs différents crédentiels
- La définition des actions requises
- L'activation éventuelle de la self-registration
- La personnalisation des attributs

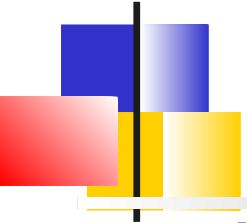
D'autre part, Keycloak peut s'intégrer avec des systèmes externes comme LDAP ou d'autres fournisseurs d'identité : "fédération d'utilisateurs"



Required User Actions

Keycloak permet d'interagir avec les utilisateurs pendant le processus d'authentification à l'aide d'une fonctionnalité appelée **Required User Actions**

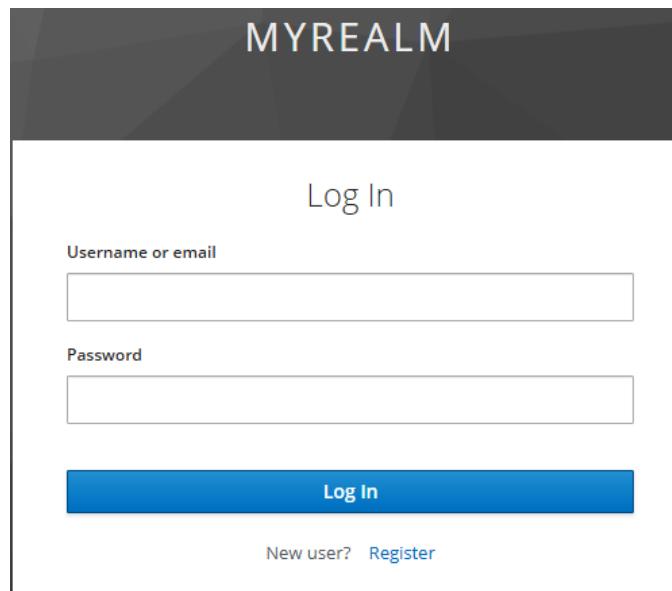
- **Verify Email** : envoyer un e-mail à l'utilisateur pour confirmer qu'il appartient à cet utilisateur.
- **Update Password**: demandez à l'utilisateur de mettre à jour son mot de passe.
- **Update Profile**: demandez à l'utilisateur de mettre à jour son profil en fournissant son prénom, son nom et son adresse e-mail.

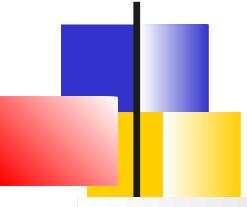


Self-registration

Keycloak permet aux utilisateurs de s'enregistrer par eux même

Realm Settings → Login → User registration



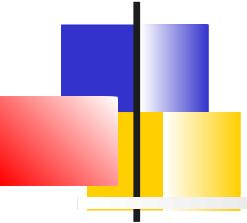


Compte utilisateurs

Empêcher un attaquant d'accéder à un compte utilisateur consiste principalement à activer une authentification forte (pas seulement un mot de passe comme moyen d'authentification).

La protection des mots de passe nécessitent :

- Algorithme de hachage de mot de passe fort
Realm Settings → Authentication → Password Policy
- Une bonne politique pour les mots de passe (combinaison de caractères spéciaux, digits, majuscule, minuscule, longueur)
Realm Settings → Authentication → Password Policy
- Une protection contre les attaques force-brute
Realm Settings → Security Defense → Brute Force Protection
- Éduquer les utilisateurs afin qu'il utilise différents mots de passe entre service



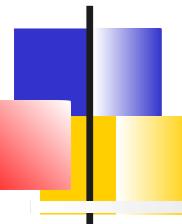
Intégration LDAP ou Active Directory

Keycloak peut s'intégrer à LDAP de 2 façons :

- Les données de l'annuaire LDAP sont importées dans la base de données Keycloak, et synchronisées
3 modes de synchronisation possible : READ_ONLY, WRITABLE et UNSYNCED
- La vérification des informations d'identification sont déléguées à LDAP.

Il est possible de configurer plusieurs annuaires LDAP au sein d'un même realm et de configurer un ordre de priorité.

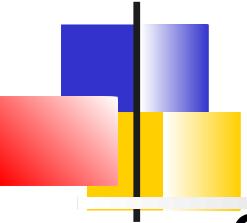
Lors de l'intégration, des mappers sont configurés pour faire correspondre les champs LDAP aux attributs des user keycloak, en particulier les groupes et les rôles



Autres fournisseurs d'identité

Keycloak peut s'intégrer à des fournisseurs d'identité tiers en utilisant SAML v2 ou OpenID Connect v1.0 ou aux « social providers » (Google, Github, ...)

A leur 1ère authentification, les utilisateurs sont importés dans Keycloak.
Ils peuvent alors profiter de toutes les fonctionnalités fournies par Keycloak et respecter les contraintes de sécurité imposées par votre royaume



Intégration avec un BD spécifique

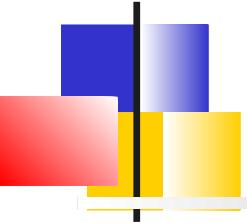
Si on veut conserver les utilisateurs dans son propre schéma relationnel !

Plusieurs alternatives :

- Implémenter un SPI en Java implémentant l'interface **UserStorageProvider**.
Déployer ce provider dans Keycloak
Les utilisateurs ne sont plus présents dans keycloak¹
- Utiliser l'API Rest de Keycloak. Et implémenter une synchronisation des users

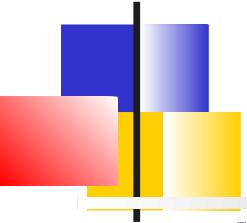
1. <https://github.com/keycloak/keycloak-quickstarts/tree/latest/extension/user-storage-jpa>





Administration Keycloak

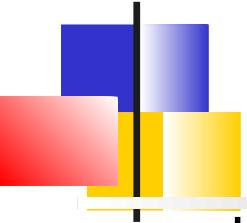
Organisation
Gestion des utilisateurs
Flow d'authentification
Gestion des sessions et jetons



Introduction

Keycloak permet de configurer l'authentification des utilisateurs :

- Politique sur les mots de passe
- Utilisation de OTP (One Time Password)
- Utilisation de WebAuthn (Web Authentication)
- Configuration des séquences de l'authentification

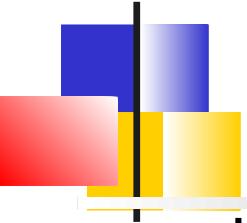


Politique des mots de passe

Keycloak permet de définir des contraintes sur les mots de passe

- Algorithme de hachage
Par défaut : PBKDF2
- Itérations de Hash
Impact sur les performances
- Digits, LowerCase, UpperCase, Caractères spéciaux
- NotUsername, Not Email
- Expression régulière
- Date d'expiration
- Liste noire : Les mots de passe interdits

Authentication → Policies → Password Policy



Flux d'authentification

Keycloak permet de configurer toutes les étapes d'un processus d'authentification (Authentication Flow)

- Le processus englobe des actions, des écrans de saisie,...
- Certains flows sont prédéfinis.
Les étapes ne peuvent pas être redéfinies, on peut juste les marquer comme obligatoire et facultative
- On peut redéfinir un processus d'authentification complet et l'associer à un cas d'usage

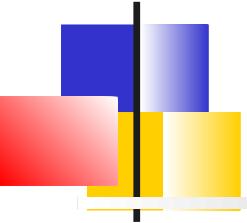
Les flows d'authentification sont définies au niveau realm, peuvent être surchargés au niveau client ou associés à des client policies.

Exemple Browser Flow

The screenshot shows the Keycloak Administration interface. The left sidebar is dark with white text, listing various management options. The 'Authentication' option is selected and highlighted in grey. The main content area is titled 'Authentication > Flow details' and shows a 'Browser' flow configuration. The flow consists of several steps:

Steps	Requirement
Cookie	Alternative
Kerberos	Disabled
Identity Provider Redirector	Alternative
forms Username, password, otp and other auth forms.	Alternative
Username Password Form	Required
Browser - Conditional OTP Flow to determine if the OTP is required for the authentication	Conditional
Condition - user configured	Required
OTP Form	Required

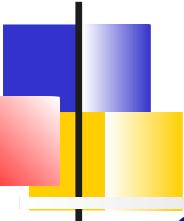
At the bottom of the page are two buttons: '+ Add step' and '+ Add sub-flow'.



Attributs des étapes

Des radio buttons contrôlent l'exécution des étapes du flow :

- **Required** : Tous les éléments requis du flux doivent être exécutés séquentiellement avec succès.
Le flux se termine si un élément requis échoue.
- **Alternative** : Un seul élément doit s'exécuter avec succès pour que le flux soit évalué comme réussi.
=> Tout élément alternative dans un flux contenant des éléments requis ne s'exécutera pas.
- **Disabled** : L'élément n'est pas pris en compte pour marquer un flux comme réussi.
- **Conditional** : Seulement sur des sous-flux
 - Contient des exécutions qui doivent correspondre à des instructions logiques.
 - Si toutes les exécutions sont évaluées comme vraies, le sous-flux conditionnel agit comme requis.
 - Si toutes les exécutions sont évaluées comme fausses, le sous-flux conditionnel agit comme désactivé.



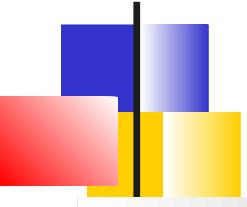
Exemple Browser Flow

Cookie : La première fois qu'un utilisateur se connecte avec succès, Keycloak définit un cookie de session. Si le cookie est déjà défini, ce type d'authentification est réussi. Étant donné que le fournisseur de cookies a renvoyé un succès et que chaque exécution à ce niveau du flux est ALTERNATIVE, Keycloak n'effectue aucune autre exécution.
=> connexion réussie.

Kerberos : Cet authenticateur est désactivé par défaut et est ignoré pendant le flux du navigateur. On peut l'activer lors de l'authentification Kerberos

Identity Provider Redirector : Redirige vers un autre fournisseur d'identité

Form : Sous-flux alternatif qui contient un type d'authentification supplémentaire qui doit être exécuté. Keycloak charge les exécutions pour ce sous-flux et les traite.



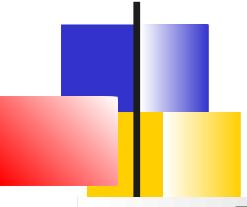
Sous-flux Form

Username Password Form : la page de login et de mot de passe. Obligatoire (REQUIRED) => nom d'utilisateur et un mot de passe valides.

Browser - Conditional OTP : Sous-flux conditionnel en fonction du résultat de condition d'exécution configuré par l'utilisateur.

Condition - User Configured authentication : Vérifie si l'utilisateur dispose d'informations d'identification OTP configurées.

OTP form : REQUIRED mais elle ne s'exécute que lorsque l'utilisateur dispose d'un identifiant OTP configuré.



OTP

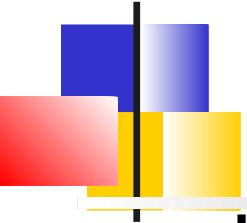
En plus de fournir un mot de passe, les utilisateurs doivent fournir une 2ème preuve de leur identité

- Un code fourni par les applications FreeOTP ou Google Authenticator disponible sur mobile
- Soit l'utilisateur, soit l'administrateur peut décider de l'utilisation de OTP

Différentes stratégies peuvent être configurées pour l'OTP :

- Time-Based One-Time Password (TOTP) : Le code est valable pendant une certaine période (30 secondes par défaut)
- HMAC-Based One-Time Password (HOTP) : Le code est valable tant qu'il n'a pas été utilisé

Authentication → Policies → OTP Policy



Mise en place OTP

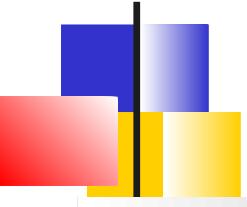
Les utilisateurs peuvent activer par eux même OTP via la console de gestion de leur compte

Signing In → Set up Authenticator Application

=> QR code représentant la clé partagée qui servira à générer les codes

Pour forcer pour tous les utilisateurs, il faut définir un autre flow d'authentification





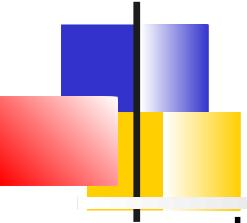
WebAuthn

WebAuthn (Web Authentication API) est une norme du W3C conçue pour remplacer les mots de passe par des méthodes d'authentification¹ plus sécurisées, comme :

- Des clés de sécurité matérielles (type YubiKey)
- L'empreinte digitale
- La reconnaissance faciale
- Le code PIN d'un appareil sécurisé

Il s'appuie sur la cryptographie à clé publique.

WebAuthn est plus sécurisé que OTP car il n'y a pas de clé partagée entre Keycloak et les applications tierces (FreeOtp et GoogleAuthenticator)



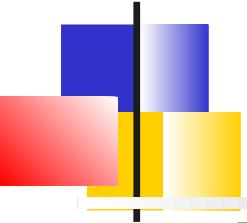
Fonctionnement général

Lors de l'enregistrement :

- L'authenticator génère une paire de clés (publique/privée).
- La clé publique est envoyée au serveur (ici Keycloak).
- La clé privée est stockée dans l'authenticator (clé matérielle, OS, etc.).

Lors de l'authentification :

- Le serveur envoie un challenge.
- Le navigateur le signe en faisant appel à l'authenticator avec la clé privée.
- Le serveur vérifie la signature avec la clé publique.

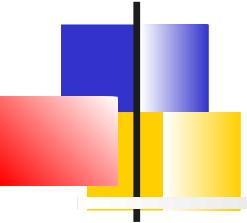


Mise en place Webauthn

Définir un flow avec
***WebAuthn authentication ou
WebAuthn Passwordless***

Dans sa console, l'utilisateur enregistrer
son Authenticator





acr

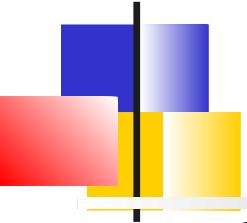
acr (Authentication Context Class Reference)

est un claim standard qui indique le niveau ou le contexte de l'authentification utilisée pour connecter un utilisateur.

Avec Keycloak, on peut associer (au niveau realm ou au niveau client) des valeurs d'acr à des entiers : les **LoA (Level Of Authentication)**

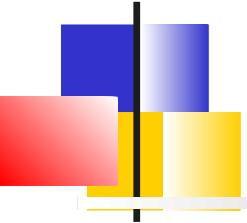
Il est alors possible de mettre au point un flow d'authentification avec une étape conditionnel dépendant du LoA





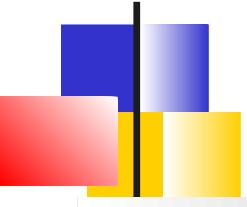
Scénario

1. Lors de la requête OIDC initiale, le client précise une valeur du paramètre `acr_values`. Ex :
`acr_values=urn:myorg:mfa`
2. Keycloak mappe cette valeur à un LoA
3. Le flow d'authentification qui s'applique dépend du LoA
4. A l'issue de l'authentification, le claim **`acr`** est dans l'ID Token



Administration Keycloak

Organisation
Gestion des utilisateurs
Flows d'authentification
Gestion des sessions et jetons



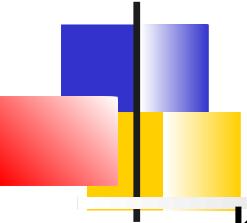
Introduction

Les sessions permettent à Keycloak de déterminer si les utilisateurs et les clients sont authentifiés, pendant combien de temps ils doivent être authentifiés et quand il est temps de les ré-authentifier.

Les sessions sont conservées en mémoire et peuvent avoir un impact sur Keycloak

L'exploitant Keycloak peut alors :

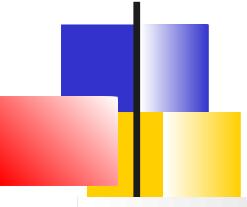
- Configurer des timeouts
- Visualiser les sessions actives
- Faire de l'audit
- Forcer la fermeture de session



Types de sessions

Keycloak maintient 2 types de session lorsqu'un utilisateur s'authentifie :

- **La session Single Sign-on (SSO)**, permet de suivre l'activité de l'utilisateur quelque soit le client/application
La configuration de ses timeouts contrôle la fréquence des ré-authentifications utilisateurs et clients
Lorsque la session expire, toutes les sessions client associées expirent
- **Les sessions cliente** Sous-sessions associées à chaque client (application) accédé par l'utilisateur.
Gère : les jetons (Access, Refresh), les scopes autorisés, les timeouts propres à ce client (si configurés).
Expire si la session sso expire ou si le refresh token associé expire ou est révoqué



Timeout

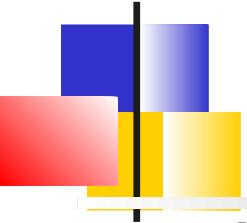
La configuration des timeout pour les différentes sessions se fait dans
Realm settings → Sessions

On définit principalement :

- ***Idle timeout*** : La durée faisant expirer la session si il n'y a pas d'activité user ou de rafraîchissement de jeton (par défaut 30mn)
- ***Max time*** : La durée maximale d'une session

Au niveau d'un client :

Client → Settings → Login Settings

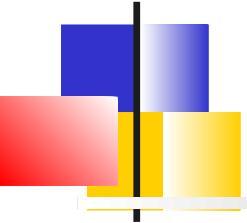


Gestion des sessions actives

Les administrateurs ont une visibilité des sessions actives au niveau

- Du realm
- D'un client
- D'un utilisateur

Il peut forcer un logout à tous les niveaux

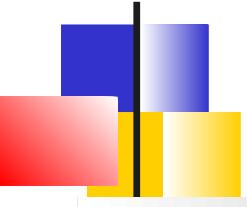


Cookies

Après une authentification réussie,
Keycloak positionne un cookie
HttpOnly¹ **KEYCLOAK_IDENTITY** avec
une expiration correspondant au Max
Time d'une session

Si https, il positionne également le flag
secure sur le cookie empêchant un
transfert en clair

1. Le cookie n'est pas accessible par le javascript du navigateur pour se prévenir des attaques XSS



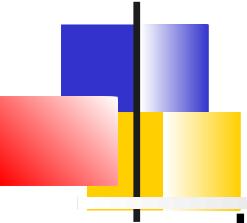
Jetons

Les jetons sont généralement liés à des sessions.

Par conséquent, la validité des jetons (pas nécessairement leur durée de vie) dépend des sessions.

Les jetons ont leur propre durée de vie et la durée pendant laquelle ils sont considérés comme valides dépend de la façon dont ils sont validés

Avec JWT, les serveurs de ressources valident indépendamment de Keycloak un jeton
=> Pendant sa durée de vie, la session peut avoir expirée
=> Si l'on ne veut pas, il faut se rabattre sur une validation en utilisant l'endpoint d'introspection

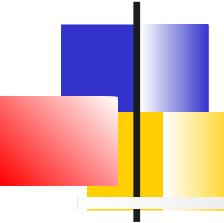


Durée de vie des jetons

Les jetons d'identification et d'accès partagent la même durée de vie, généralement courte car ils sont utilisés par des clients publics et fréquemment échangés

Les jetons de rafraîchissement ont une durée plus longue mais leur validité dépend de la durée de vie définie pour les sessions utilisateur et client

Dans Keycloak, leur durée de vie suivent la configuration des timeout idle des sessions SSO ou sessions clientes



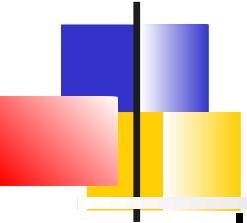
Configuration des jetons

La configuration s'effectue dans

Realm Settings → Tokens

Les valeurs par défaut sont 5 minutes pour les jetons d'accès et d'identification

On peut configurer globalement ou par clients



Jetons de rafraîchissement

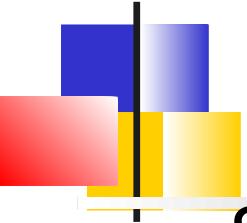
Ils sont toujours liés à une session client

Ils sont considérés comme valides si les sessions utilisateur et client auxquelles ils sont liés n'ont pas expiré.

Les clients peuvent utiliser des jetons d'actualisation pour obtenir de nouveaux jetons uniquement si leurs sessions client respectives sont toujours actives.

Ils offrent une surface plus longue aux attaques

=> un client confidentiel a généralement des jetons d'actualisation plus long que les clients publics



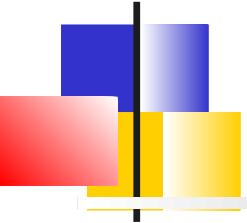
Rotation des jetons de rafraîchissement

Stratégie pour empêcher les attaquants de réutiliser les jetons d'actualisation consistant à invalider le jeton avant d'en émettre un nouveau

- Cela permet d'identifier rapidement le moment où le jeton d'actualisation a fuité et forcer l'attaquant ou le client légitime à se ré-authentifier

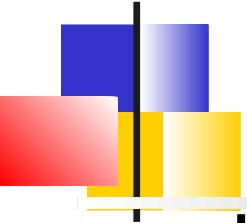
En activant cette stratégie, l'administrateur doit définir également

- **Refresh Token Max Reuse** : qui définit combien de fois un jeton de rafraîchissement peut être réutilisé. Par défaut 1



Développement serveur

Admin API
Thème
SPI

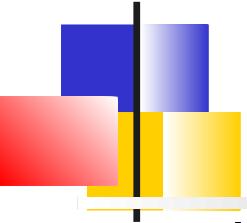


Introduction

Keycloak propose une API Rest d'administration

Pour pouvoir l'invoquer, il faut obtenir un jeton avec les permissions adéquates

- En utilisant le client **admin-cli** et le login d'un compte administrateur
- En créant, un client spécifique qui inclut dans son token d'accès une audience **security-admin-console** et un rôle **admin¹**



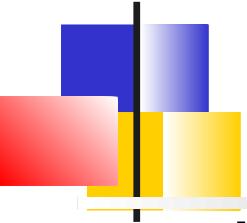
Obtention du jeton

Jeton via l'admin-cli

```
curl \  
-d "client_id=admin-cli" \  
-d "username=admin" \  
-d "password=password" \  
-d "grant_type=password" \  
"http://localhost:8080/realmss/master/protocol/openid-connect/token"
```

Jeton via client credentials après avoir configuré un Audience Mapper

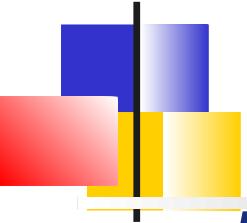
```
curl \  
-d "client_id=<YOUR_CLIENT_ID>" \  
-d "client_secret=<YOUR_CLIENT_SECRET>" \  
-d "grant_type=client_credentials" \  
"http://localhost:8080/realmss/master/protocol/openid-connect/token"
```



Admin API

L'API permet de :

- Manipuler les realms
- Manipuler les users / groupes /realm roles
- Les clients/ client scopes / rôles associés aux clients
- Les mappers
- ...
- Mais également certains endpoint pour détecter des attaques brute force



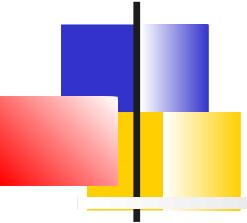
kcadm.sh

kcadm.sh est l'outil en ligne de commande officiel pour administrer Keycloak via son API REST.

Il permet de gérer les utilisateurs, rôles, clients, groupes, etc.

Utile pour automatiser les tâches d'administration dans des scripts ou pipelines CI/CD.

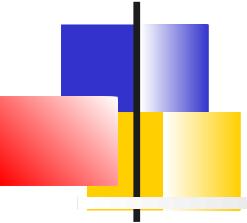
Livré avec chaque distribution Keycloak (présent dans /bin).



Authentification

Avant l'utilisation de l'API rest, se connecter à l'instance Keycloak

```
kcadm.sh config credentials \
  --server http://localhost:8080 \
  --realm master \
  --user admin \
  --password admin
```



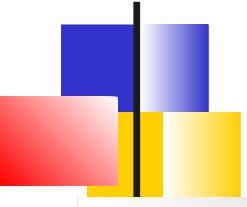
Quelques commandes

```
# Lister les utilisateurs du realm 'myrealm'  
kadm.sh get users -r myrealm
```

```
# Créer un utilisateur  
kadm.sh create users -r myrealm \  
-s username=testuser -s enabled=true
```

```
# Ajouter un rôle à un utilisateur  
kadm.sh add-roles -r myrealm \  
--username testuser \  
--rolename user
```

Possibilité de chainer des appels et de manipuler des IDs via *jq* ou *grep*.



Fichiers json

Importation d'un realm

```
kcadm.sh create realms -f realm-export.json
```

Création d'un user

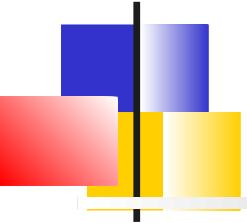
```
kcadm.sh create users -r monrealm -f user.json
```

```
kcadm.sh set-password -r monrealm --userid  
$USER_ID --new-password monpass123
```

Création d'un client

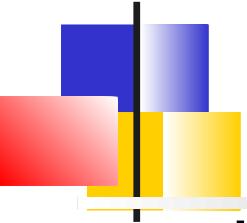
```
kcadm.sh create clients -r monrealm -f client.json
```





Développement serveur

Admin API
Thème
SPI

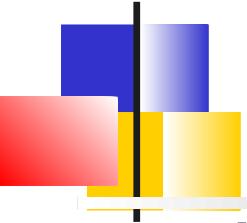


Introduction

Keycloak permet de personnaliser les pages utilisés par les utilisateurs via des thèmes :

- Formulaires de login
- Pages de l'application account
- Email
- Pages d'accueil Keycloak
- Console d'administration

Il existe des thèmes prédéfinis et il est possible de développer son propre thème



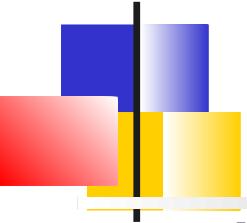
Configuration

La configuration s'effectue via la console d'admin

Realm Settings → Theme

Un thème peut également être précisé au démarrage du serveur :

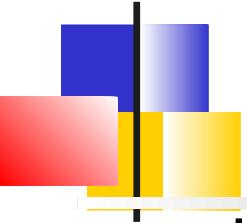
```
bin/kc.[sh|bat] start --spi-theme-welcome-  
theme=custom-theme
```



Eléments d'un thème

Un thème est constitué de :

- De gabarits freemarker
- D'images
- De messages localisés (bundle)
- De feuilles de style css
- De script
- De propriétés



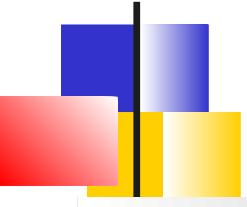
Héritage

La création s'effectue généralement en étendant :

- Le thème **Keycloak** qui contient des images et les feuilles de style si l'on veut modifier les gabarits (ajout de lien par exemple)
- Ou le thème **base** qui contient les gabarits et les messages si l'on veut travailler le look and feel

Après avoir étendu un thème on peut surcharger ses éléments

Lors de la création de thème, il est recommandé de désactiver la fonctionnalité de cache pour éviter de redémarrer le serveur



Procédure

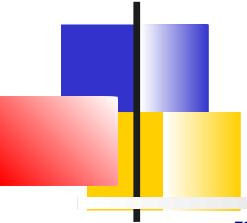
1) Désactivation du cache :

```
bin/kc.[sh|bat] start --spi-theme-static-max-  
age=-1 --spi-theme-cache-themes=false --spi-theme-  
cache-templates=false
```

2) Création d'un sous-répertoire dans le répertoire
themes

3) Puis création de répertoire pour chaque type de
thème (ex. /login pour la page de login)

4) Pour chaque type créer un fichier
theme.properties



Propriétés de thème

parent : Le thème parent à étendre

import : Importer des ressources d'un autre thème

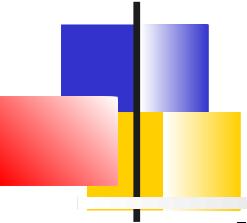
common : Surcharge le chemin vers les ressources communes. (par défaut est `common/keycloak`) Utilisé dans les gabarits *Freemarker* via `${url.resourcesCommonPath}`

styles/scripts : Listes de styles/script à inclure. Les fichiers sont placés dans `/resources/css` | `/resources/js` du répertoire du thème

locales : Liste des langues supportés

Des propriétés sont également utilisées pour changer les classes css de certains éléments

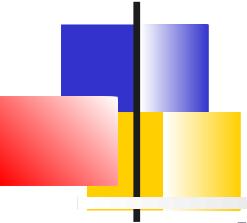
Il est également possible de définir des propriétés custom qui seront utilisées dans les gabarits et d'utiliser des propriétés JVM ou de l'environnement



Création de gabarit

Il est possible de surcharger un gabarit du parent en plaçant un gabarit freemarker à la racine du répertoire

La liste des gabarits que l'on peut surcharger est visible en ouvrant le jar lib/lib/main/org.keycloak.keycloak-themes-22.0.5.jar présent dans la distribution

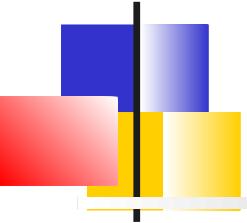


Déploiement du thème

Les thèmes peuvent être déployés

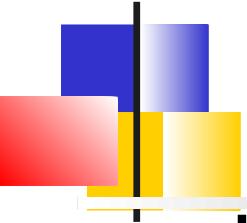
- En copiant le répertoire de thème dans le répertoire *themes* de la distribution
- En créant une archive qui contient les ressources + un fichier *META-INF/keycloak-themes.json* listant les fichiers ressources du jar et en plaçant l'archive dans le répertoire *providers* de la distribution





Développement serveur

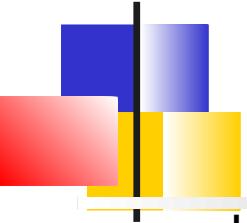
Admin API
Thème
SPI



Introduction

Les **SPI (Service Provider Interfaces)** sont des extensions personnalisées qui permettent aux développeurs d'étendre, de personnaliser et d'intégrer Keycloak avec d'autres systèmes ou services.

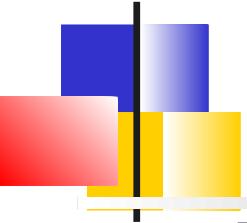
Les SPI offrent une flexibilité et une extensibilité considérables pour adapter Keycloak à des cas d'utilisation spécifiques



SPIs disponibles

Les principales SPIs sont :

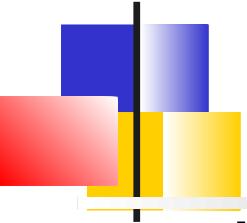
- **Authenticator SPI** : Étendre les flows d'authentification
- **User Storage SPI** : Stocker les utilisateurs dans un système de persistance personnalisé
- **Federated Identity Provider SPI** : Pour intégrer des fournisseurs d'identité externes
- **Protocol Mapper SPI** : Pour ajouter des informations personnalisés dans les jetons
- **Event Listener SPI** : Pour intégrer des listeners des événements Keycloak
- **Action Token Handler SPI** : Des jetons permettant des actions côté Keycloak (Réinitialiser les mots de passe, Confirmer une adresse e-mail, Executer des required action(s))
- **Vault SPI** : Connexion à une implémentation Vault
- ...



Implémentation

Pour implémenter une SPI, il faut :

- Implémenter 2 interfaces
ProviderFactory et ***Provider***
- Créer un fichier de configuration du service qui référence la factory

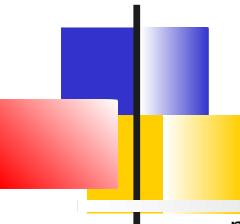


Interface factory

Le serveur *Keycloak* instancie la factory comme singleton et appelle ses méthodes de callback *init()*, *postInit()*, *close()*.

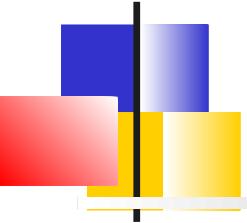
La responsabilité de la factory est de créer l'implémentation du provider lors de sa méthode *create()* et de fournir un identifiant :

- Unique
- Ou surchargeant un service existant



Exemple *ThemeSelectorProviderFactory*

```
public class MyThemeSelectorProviderFactory implements ThemeSelectorProviderFactory {  
  
    @Override  
    public ThemeSelectorProvider create(KeycloakSession session) {  
        return new MyThemeSelectorProvider(session);  
    }  
  
    @Override  
    public void init(Config.Scope config) {  
    }  
  
    @Override  
    public void postInit(KeycloakSessionFactory factory) {  
    }  
  
    @Override  
    public void close() {  
    }  
  
    @Override  
    public String getId() {  
        return "myThemeSelector";  
    }  
}
```



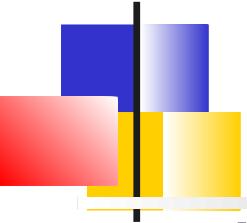
Exemple *ThemeSelectorProvider*

```
public class MyThemeSelectorProvider implements
    ThemeSelectorProvider {

    private KeycloakSession session;

    public MyThemeSelectorProvider(KeycloakSession
        session) {
        this.session = session;
    }

    @Override
    public String getThemeName(Theme.Type type) {
        return
    session.getContext().getRealm().getLoginTheme();
    }
}
```



Fichier de configuration

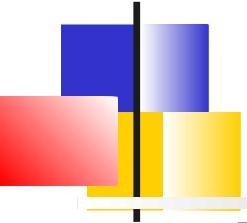
Le fichier de configuration se place
META-INF/services/ et se nomme en
fonction de la SPI

Par ex :

META-INF/services/org.keycloak.theme.ThemeSelectorProviderFactory

Il référence l'implémentation :

com.acme.provider.MyThemeSelectorProviderFactory

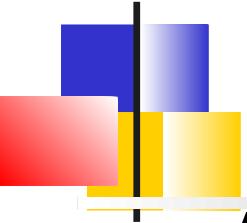


Déploiement

Dans la version *Jboss*, déployer le jar dans *standalone/deployments*

Dans la version quarkus :

- Placer le jar dans le répertoire *provider*
- Puis effectuer une commande build :
`./bin/kc.sh build`

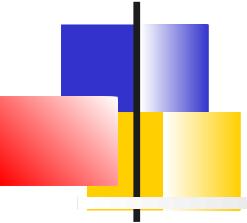


Affichage dans la console

Afin que la SPI développée s'affiche dans la page server info de la console d'administration, il faut implémenter l'interface **ServerInfoAwareProviderFactory** dans la classe factory

```
public class MyThemeSelectorProviderFactory implements  
ThemeSelectorProviderFactory, ServerInfoAwareProviderFactory {  
    ...  
  
    @Override  
    public Map<String, String> getOperationalInfo() {  
        Map<String, String> ret = new LinkedHashMap<>();  
        ret.put("theme-name", "my-theme");  
        return ret;  
    }  
}
```

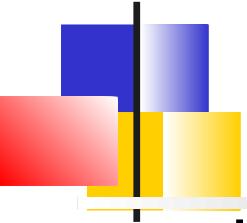




Vers la production

Configuration et optimisation du démarrage

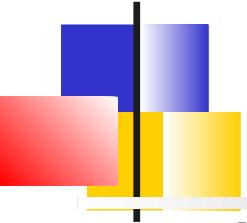
Configuration de production
Sécurisation Keycloak



Sources de configuration

Keycloak charge sa configuration à partir de quatre sources, par ordre de précédence :

- Paramètres de ligne de commande
 - <key-with-dashes>=<value>
 - ex : --db-url=<value>
- Variables d'environnement
 - KC_<key_with_underscores>=<value>
 - ex : KC_DB_URL=<value>
- Fichier *.conf* créé par l'utilisateur
- Fichier *keycloak.conf* situé dans le répertoire conf.
 - <key-with-dashes>=<value>
 - ex : db-url=<value>



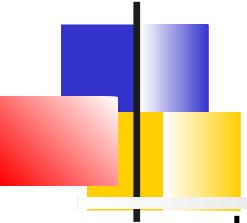
Fichiers de configuration

Les fichiers de configuration peuvent faire référence à des variables d'environnement et à des valeurs par défaut :

```
db-url-host=${MY_DB_HOST:mydb}
```

Un fichier de configuration dédié peut être fourni au démarrage :

```
bin/kc.[sh|bat] --config-file=/path/to/myconfig.conf start
```



Démarrage

Keycloak peut être démarré en 2 modes :

Démarrage en mode dev

bin/kc.[sh|bat] start-dev

HTTP est activé

La résolution stricte du nom d'hôte est désactivée

Le cache est défini sur *local*

La mise en cache des thèmes et des modèles est désactivée (utile pour le développement de thème)

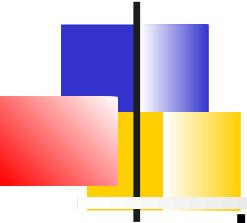
Démarrage en mode production

bin/kc.[sh|bat] start

HTTP est désactivé

Configuration du Hostname requise

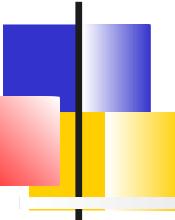
HTTPS/TLS requis



Utilisateurs admin

Lors du 1^{er} démarrage, Keycloak utilise les variables d'environnement `KEYCLOAK_ADMIN` et `KEYCLOAK_ADMIN_PASSWORD` pour créer un utilisateur avec les droits d'administration.

Ensuite, on peut utiliser la console d'administration ou le script `kcadm` pour créer des utilisateurs supplémentaires

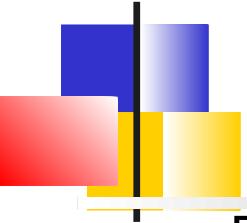


Optimisation du démarrage

Il est recommandé d'optimiser Keycloak pour de meilleurs temps de démarrage et une meilleure consommation de mémoire avant de déployer dans des environnements de production.

Lors de l'utilisation des commandes *start* ou *start-dev*, Keycloak¹ exécute une commande de **build** qui peut prendre du temps.

=> Pour optimiser, exécuter cette commande de build avant le démarrage en fixant la plupart des valeurs de configuration, mettre au point un fichier *keycloak.conf* contenant les clés de configuration définies au runtime puis démarrer avec l'option **optimized**.



Exemple optimisation

Phase de build

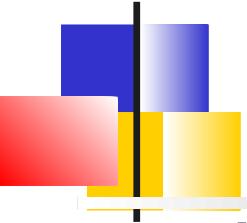
```
bin/kc.[sh|bat] build --db=postgres
```

Keycloak.conf :

```
db-url-host=keycloak-postgres  
db-username=keycloak  
db-password=change_me  
hostname=mykeycloak.acme.com  
https-certificate-file
```

Démarrage du serveur

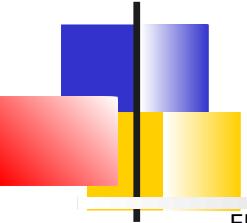
```
bin/kc.[sh|bat] start --optimized
```



Cas d'un container

L'image du conteneur Keycloak par défaut est livrée prête à être configurée et optimisée.

On peut donc inclure la phase de build de Keycloak durant la construction du conteneur



Exemple Dockerfile

```
FROM quay.io/keycloak/keycloak:latest as builder

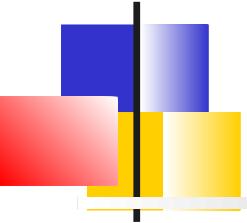
# Enable health and metrics support
ENV KC_HEALTH_ENABLED=true
ENV KC_METRICS_ENABLED=true

# Configure a database vendor
ENV KC_DB=postgres

WORKDIR /opt/keycloak
# for demonstration purposes only, please make sure to use proper certificates in production instead
RUN keytool -genkeypair -storepass password -storetype PKCS12 -keyalg RSA -keysize 2048 -dname "CN=server" -alias server -ext "SAN:c=DNS:localhost,IP:127.0.0.1" -keystore conf/server.keystore
RUN /opt/keycloak/bin/kc.sh build

FROM quay.io/keycloak/keycloak:latest
COPY --from=builder /opt/keycloak/ /opt/keycloak/

# change these values to point to a running postgres instance
ENV KC_DB_URL=<DBURL>
ENV KC_DB_USERNAME=<DBUSERNAME>
ENV KC_DB_PASSWORD=<DBPASSWORD>
ENV KC_HOSTNAME=localhost
ENTRYPOINT ["/opt/keycloak/bin/kc.sh"]
```

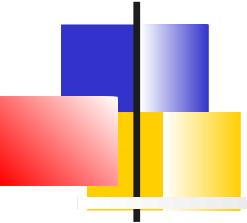


Build et démarrage

Après avoir mis au point le fichier
Dockerfile

```
podman|docker build . -t mykeycloak
```

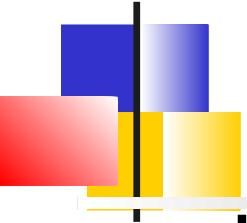
```
podman|docker run --name mykeycloak -p 8443:8443 \
-e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=change_me \
mykeycloak \
start --optimized
```



Vers la production

Configuration et optimisation du
démarrage

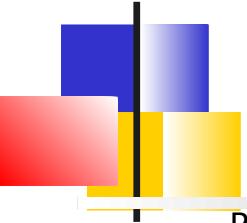
Configuration de production
Sécurisation Keycloak



Configuration de production

La configuration de production nécessite :

- De configurer **TLS** pour les communications http
- Configurer le **hostname**
- Mettre en place un **reverse proxy**
- Mettre en place le **clustering**
- Configurer une **base de données de production**



TLS

Pour charger les certificats, Keycloak s'appuie sur des fichiers PEM ou des keystores

PEM

```
bin/kc.[sh|bat] start --https-certificate-file=/path/to/certfile.pem  
--https-certificate-key-file=/path/to/keyfile.pem
```

Keystore

L'emplacement par défaut est conf/keystore

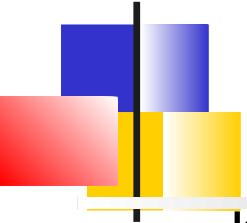
```
bin/kc.[sh|bat] start -https-key-store-password=<value>
```

Keycloak utilise le port 8443, cela peut être changé avec :

```
bin/kc.[sh|bat] start -https-port=<port>
```

L'authentification du client est également supportée :

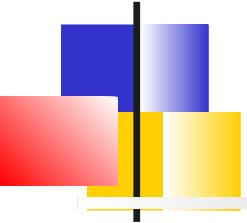
```
bin/kc.[sh|bat] start --https-trust-store-file=/path/to/file  
--https-trust-store-password=<value>  
--https-client-auth=<none|request|required>
```



Types d'endpoints et hostname

Keycloak expose différents type d'endpoints :

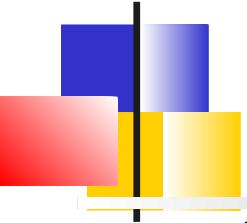
- **Frontend** : Accessibles via un domaine public et généralement liés aux flux qui passent par le canal front.
Par exemple, le *authorization_endpoint*
Clé de configuration : **hostname**
- **Backend** : Accessible via un domaine public ou un réseau privé.
Ils sont utilisés pour une communication directe entre le serveur et les clients.
Par exemple : *Token introspection*, *User info*, *Token endpoint*, *JWKS*
Clé de configuration : **hostname-strict-backchannel** :
 - **true** : Ces endpoints utiliseront l'adresse public
 - **false** : Ces endpoints utilisent l'adresse privé d'écoute
- **Console d'administration** : Généralement la console d'administration n'est pas accessible publiquement.
Clé de configuration : **hostname-admin**



Reverse proxy

Avec un proxy, la clé de configuration proxy doit être renseignée au démarrage :

- **edge** : Autorise la communication via HTTP entre le proxy et Keycloak.
=> réseau interne hautement sécurisé
- **reencrypt** : Communication https entre proxy et Keycloak. Différentes clés et certificats sont utilisés
- **passthrough** : le proxy transmet les requêtes à Keycloak. Les connexions sécurisées entre le serveur et les clients sont basées sur les clés et les certificats de Keycloak.



Configuration BD

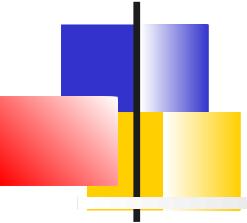
Options basiques :

```
bin/kc.[sh|bat] start  
--db postgres --db-url-host mypostgres  
--db-username myuser --db-password change_me
```

Le schéma par défaut est *keycloak*, peut être modifié via *db-schema*.

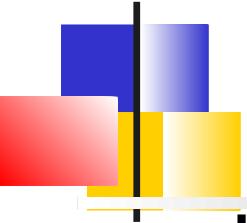
Bases supportées : *mariadb*, *mssql*, *mysql*, *oracle*, *postgres*





Vers la production

Configuration et optimisation du démarrage
Configuration de production
Clustering
Sécurisation Keycloak

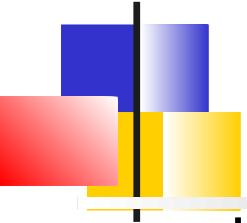


Introduction

Le clustering apporte :

- Haute disponibilité (HA)
- Scalabilité horizontale
- Réplication de sessions (SSO)
- Synchronisation du cache utilisateur, tokens, sessions

Keycloak quarkus s'appuie sur 2 sous-projet Redhat indépendant JGroups et Infinispan



JGroups

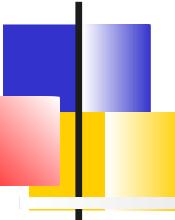
Librairie Java de communication réseau peer-to-peer

Utilisée pour la découverte des nœuds et le transport des messages

S'appuie sur différents protocoles de transport : UDP, TCP, TCPPING, KUBE_PING, etc.

Fonctions clés :

- Formation du cluster et découverte des membres
- Détection de partition réseau
- RéPLICATION des messages de cache



Infinispan - Cache distribué

Moteur de cache mémoire hautes performances

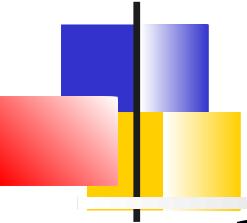
Fournit un cache distribué et répliqué

Utilisé par Keycloak pour :

- Sessions utilisateur
- Tokens
- Métadonnées clients et utilisateurs

Propose différents types de cache :

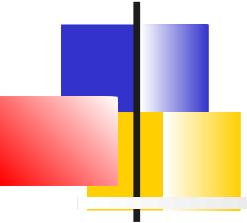
- local (non répliqué)
- répliqué (synchronisation entre tous les nœuds)
- distribué (partitionné entre nœuds)



Flux de synchronisation dans un cluster

- 1) L'utilisateur s'authentifie sur un nœud
- 2) La session est stockée dans le cache Infinispan
- 3) JGroups transmet la mise à jour aux autres nœuds
- 4) Les autres nœuds peuvent répondre à l'utilisateur sans redirection

La Sticky session n'est pas requise si le cache est bien configuré.

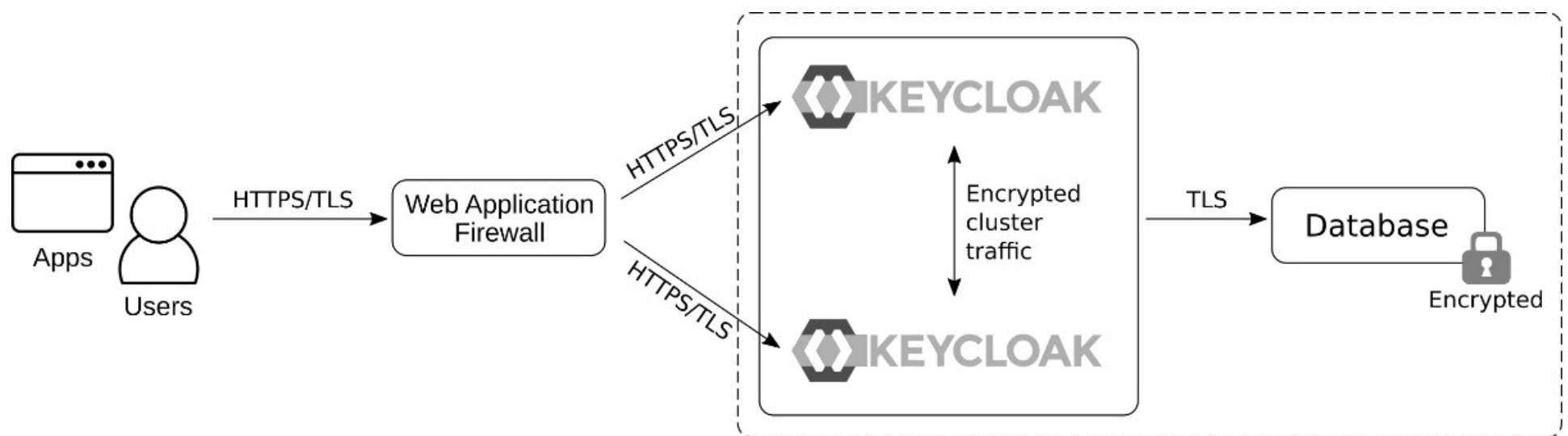


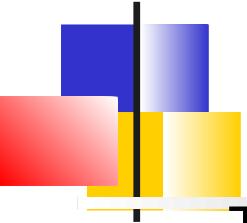
Vers la production

Configuration et optimisation du
démarrage

Configuration de production
Sécurisation Keycloak

Architecture





Sécurisation Keycloak

TLS end 2 end

- Utiliser la dernière version de TLS (TLS 1.3) et s'assurer que les librairies d'intégration choisies le supporte
- Configuration de Keycloak et du proxy en TLS passthrough

Configurer le hostname,

- sinon Keycloak détermine le hostname à partir de l'entête HTTP Host
=> Un attaquant peut facilement le tromper

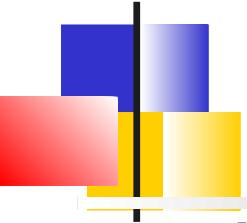
Rotation des clés de signature utilisées par Keycloak

- Realm Settings → Keys : Par exemple 1 fois par mois

Mettre à jour régulièrement Keycloak

Chargement des secrets utilisés par Keycloak à partir d'un vault externe

Protéger Keycloak avec un pare-feu et un système de prévention d'intrusion

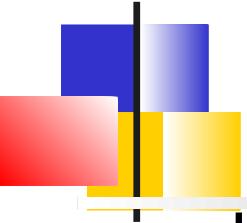


Sécuriser la BD

Keycloak stocke beaucoup de données sensibles dans sa base de données
On doit sécuriser la BD et ses backups

Recommandations minimales :

- Protéger la bd avec un pare-feu
- Activer l'authentification et le contrôle d'accès
- Crypter les backups



Cluster

Les informations concernant les sessions utilisateur sont conservées en mémoire.

Lors d'une architecture en cluster, les nœuds répliquent ses informations (InfiniSpan)

Même si ses informations ne sont pas aussi sensibles que celles de la base de données, il est recommandé de sécuriser ces échanges entre nœud Keycloak

- Autoriser l'authentification. Les nœuds doivent s'authentifier pour faire partie du cluster.
- Crypter les communications : Configuration JGroups