

Gestion centralisée de la sécurité avec KeyCloak

David THIBAU – 2023

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

Authentification avec OpenIdConnect

- Discovery, Authentification, 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, REST APIs et services

Intégration Keycloak

- Bibliothèques et adaptateurs, SpringBoot, Quarkus, Reverse Proxy

Stratégies d'autorisation

- RBAC, GBAC, OAuth2 scopes, *Authorization service*

Administration Keycloak

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

Vers 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 dédié aux applications modernes (SPA, Mobile, Restful) :

=> 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)
- Annuaire 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'authentification : 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* \Leftrightarrow 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 à tous les 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
- Le flow d'interaction avec Keycloak :
Authorization Code, Client Credentials, Device Flow
- Son usage ou pas du service d'autorisation de keycloak
- Les scopes et les rôles auxquels, elle a accès.



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 également 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

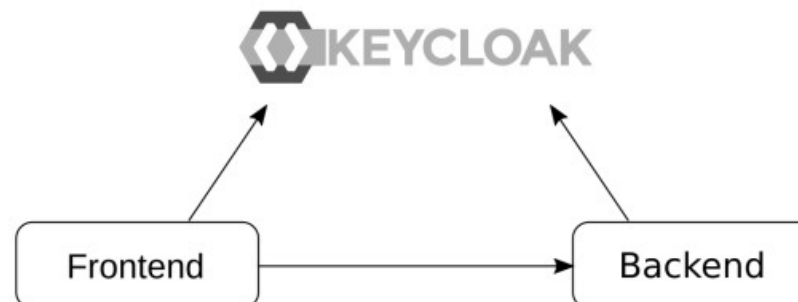
Fonctionnalités et installation
Interfaces Admin et utilisateur
Sécuriser une 1ère application



Exemple : SPA + API Rest

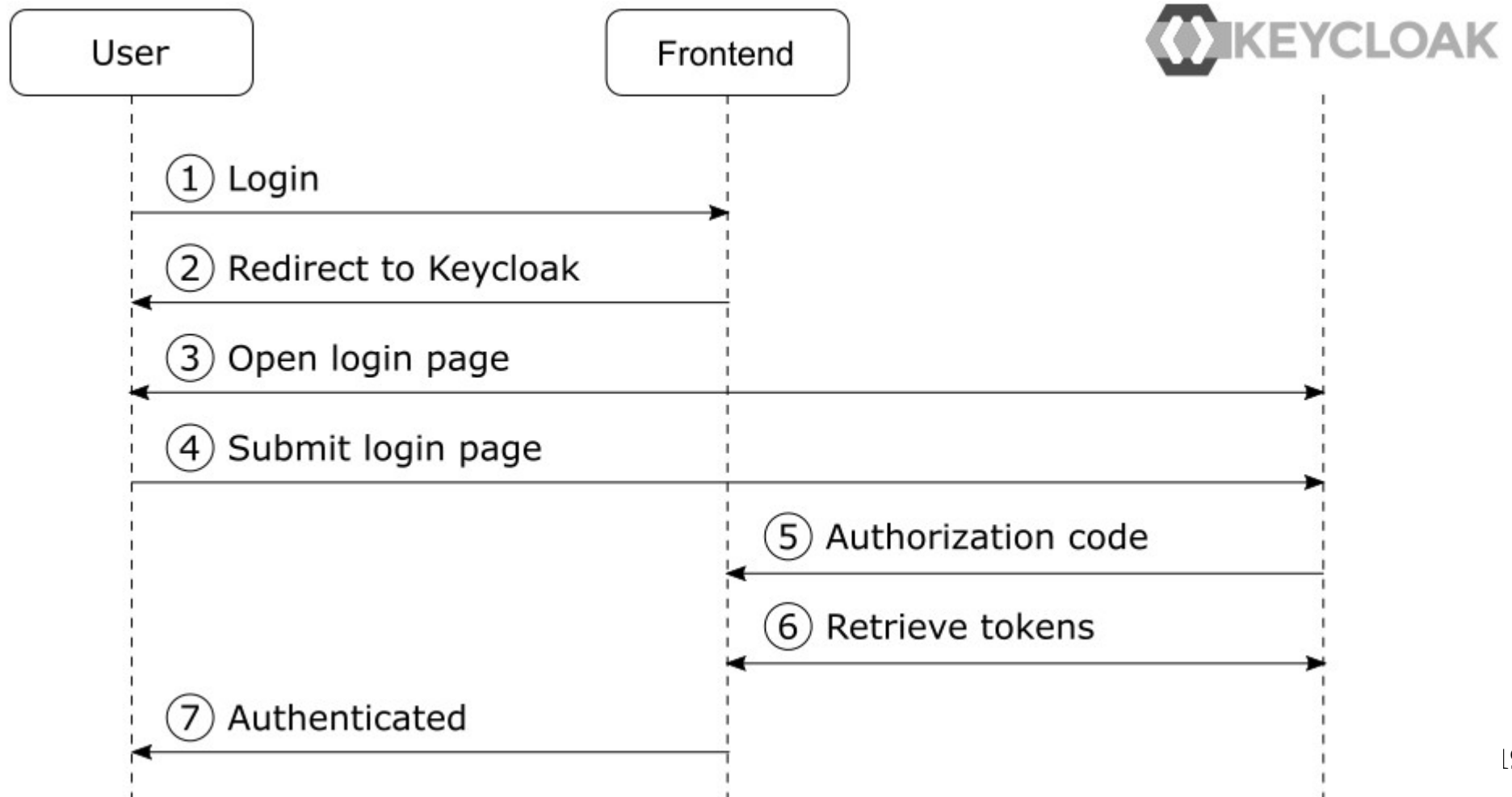
L'application exemple est composée de 2 parties :

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



Authentication FRONT-END

OpenID / Authorization code flow





Jetons

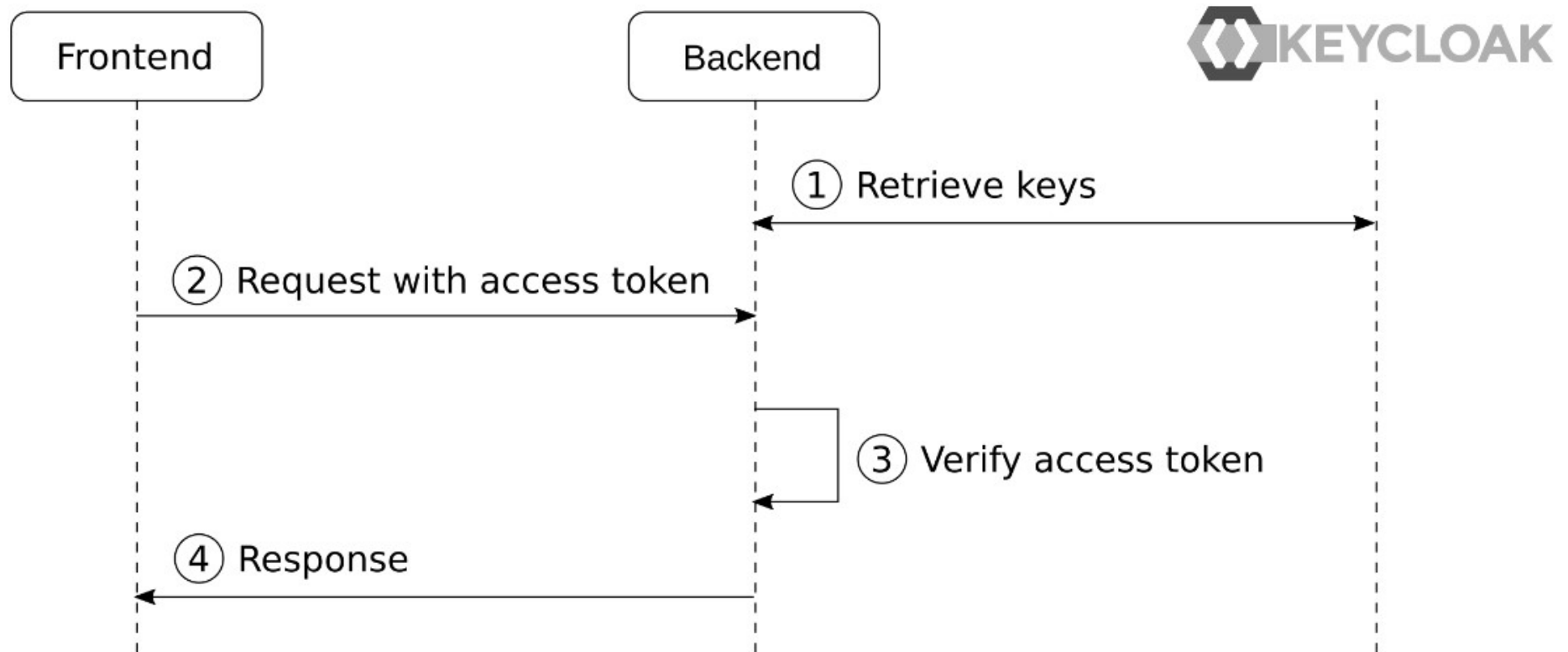
Le flow OpenId permet à l'application d'obtenir des jetons auprès de KeyCloak :

- **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** : Utilisé pour pour décider quels champs (ou revendications) inclure dans le jeton

Les jetons sont hautement personnalisables. Il existe des efforts de standardisation (MP-JWT)

Vérification des accès BACKEND

JSON Web Signature (JWS)





Rappels sur les standards

OAuth2
OpenID Connect
JWT



Apports de OAuth2

Protocole d'autorisation défini pour

- Le partage de données utilisateur entre applications
- Permet de contrôler les données partagées.

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



Rôles définis dans *OAuth2*

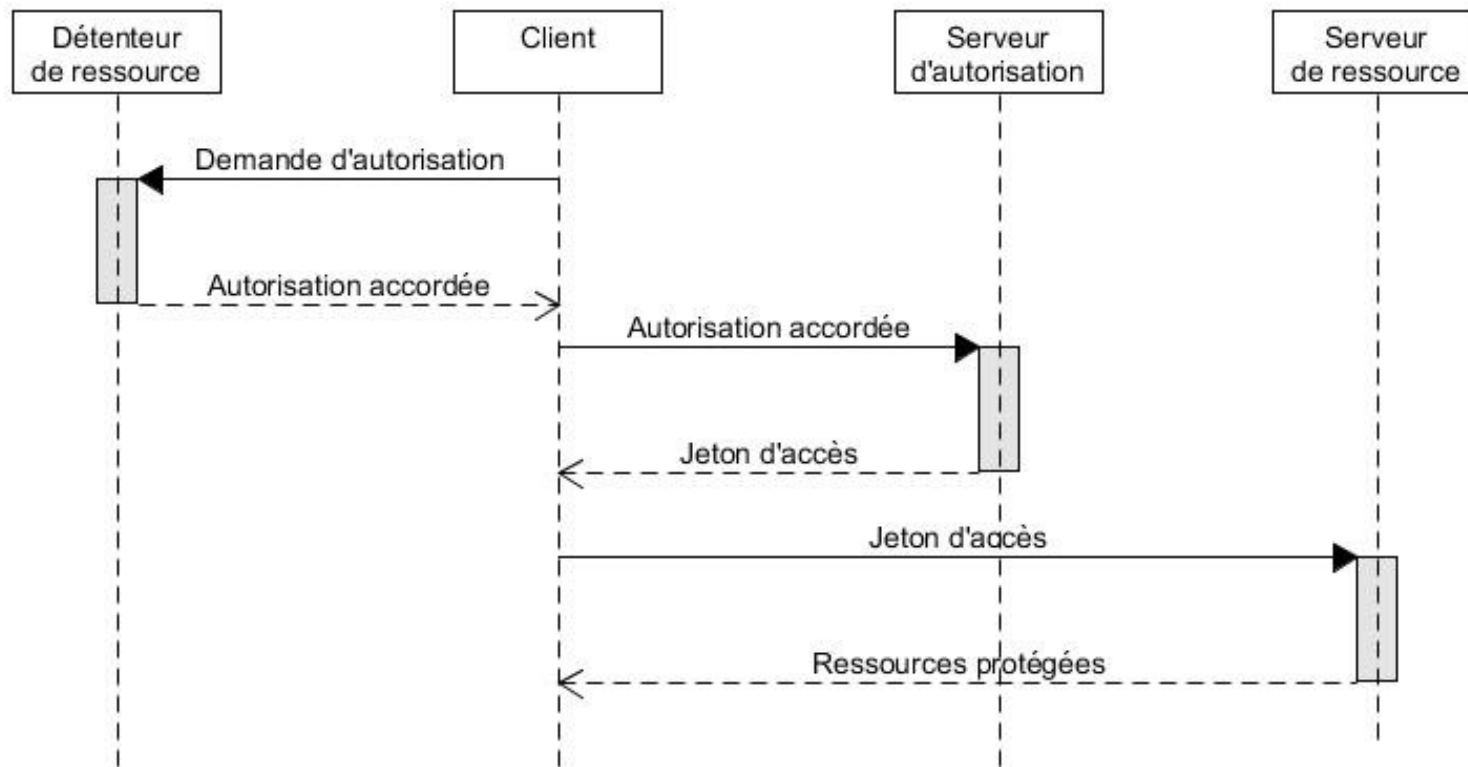
Propriétaire de la ressource : il s'agit généralement de l'utilisateur final qui possède les ressources auxquelles l'application veut accéder.

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 cas d'utilisation, il existe différentes façon d'obtenir le consentement et l'utilisateur et le jeton (Les flux d'accord):

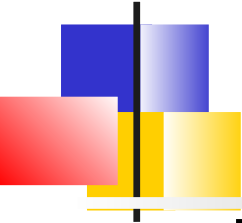
- **Authorization Code** : C'est le plus utilisé, l'utilisateur consent et est authentifié chez le fournisseur de jeton. Un code est alors renvoyé à l'application client qui l'utilise pour obtenir le jeton
- **Client Credentials** : L'application cliente détient un secret qui lui suffit pour obtenir le jeton. L'application accède à la ressource en son propre nom (l'application est le détenteur de la ressource), utilisez le .
- **Device Flow** : L'application s'exécute sur un appareil sans browser et ne peut pas détenir un secret
- Le **Password flow** est déprécié et n'est donc pas recommandé et à éviter. Il implique que l'application client obtienne les mots de passe de l'utilisateur
- **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



Types de clients

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

- Les clients **confidentiels** sont des applications pouvant stocker des créidentiels 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 créidentiels.
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.



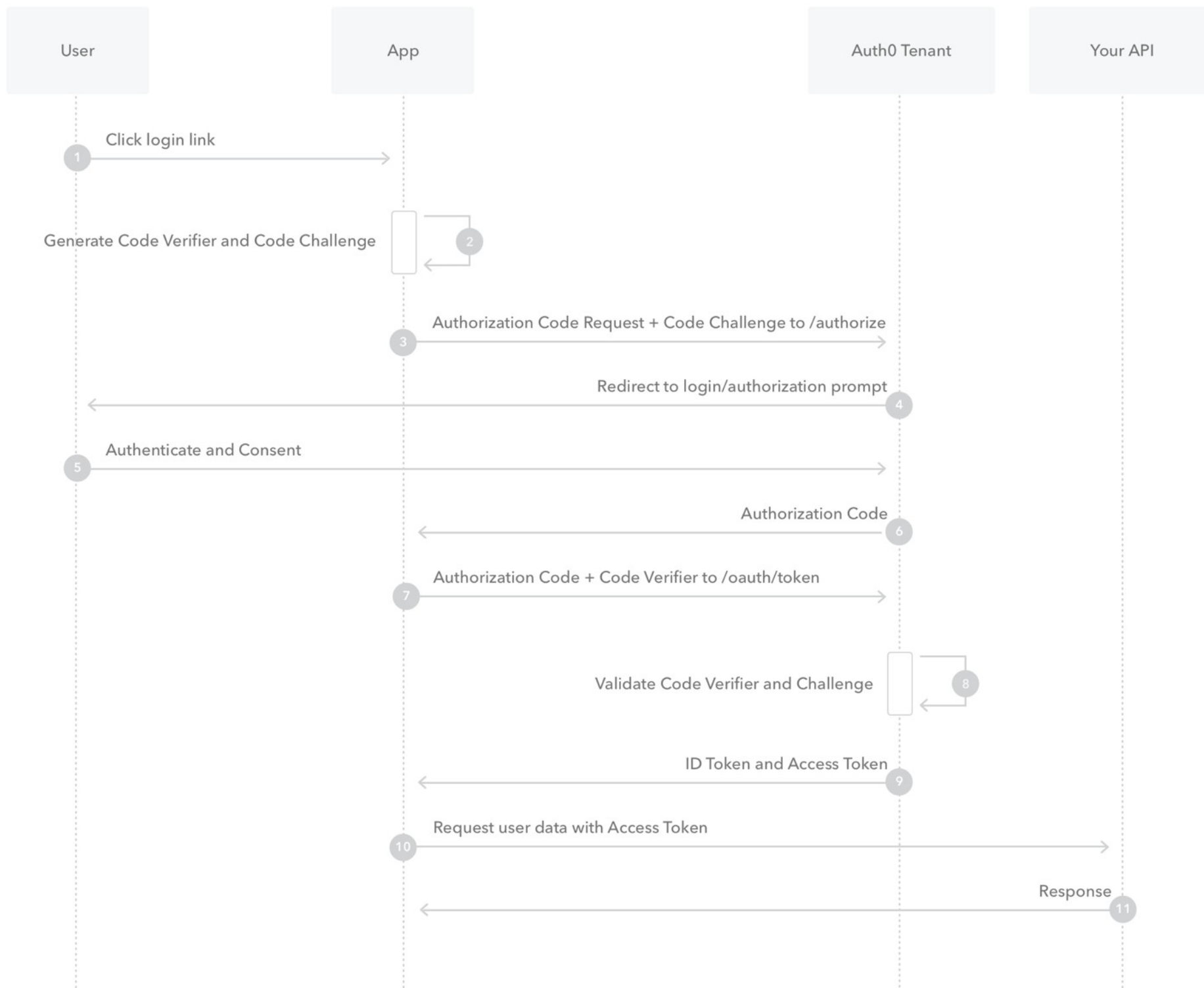
Flux avec PKCE

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/public

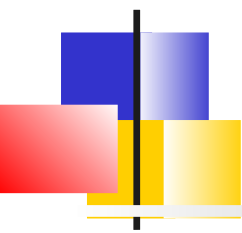


Spécifications additionnelles

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



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*

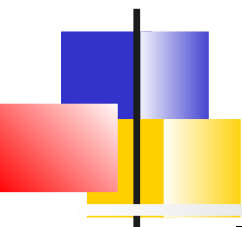


Rôles de OpenID

- **Utilisateur final** : Equivalent 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 2 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.



Spécifications additionnelles

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

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

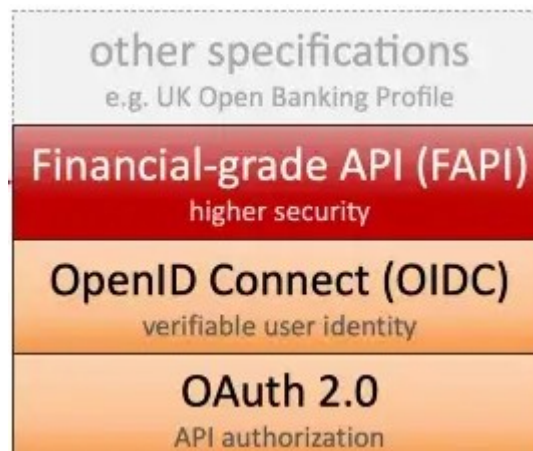
OpenID définit également un **userinfo endpoint** qui peut être accédé avec un jeton d'accès.

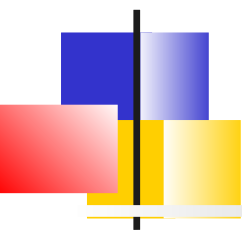
- Le endpoint fournit les mêmes informations que le jeton d'identification



Financial Grade API

Financial-grade API (FAPI) est une spécification technique qui définit des exigences techniques supplémentaires pour le secteur financier et d'autres secteurs nécessitant une sécurité plus élevée





Rappels sur les standards

OAuth2
OpenID Connect
JWT



Introduction

Keycloak utilise **JWT** comme format 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"}
```

PAYLOAD: DATA

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

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
    
) ☐ secret base64 encoded
```

✔ Signature Verified

SHARE JWT



Keycloak et OpenID

Discovery endpoint

Authentication

Jeton d'identification et userinfo
endpoint

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",
```



Authentication avec OpenID

Discovery endpoint

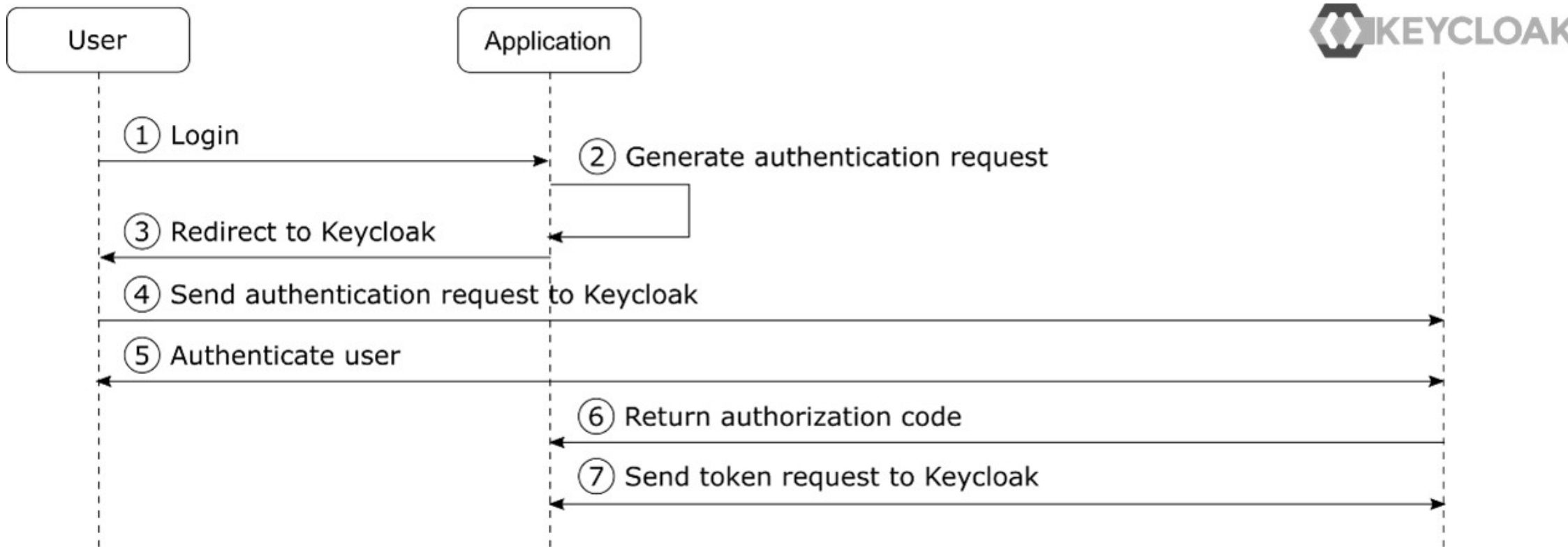
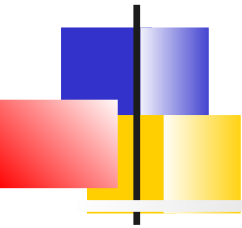
Authentication

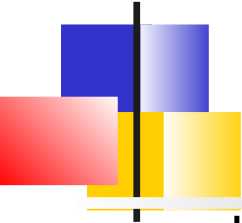
Jeton d'identification et userinfo
endpoint

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



Authentication avec OpenID

Discovery endpoint
Authentication

**Jeton d'identification et userinfo
endpoint**

Personnalisation du jeton
Logout



Pré-requis

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.

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



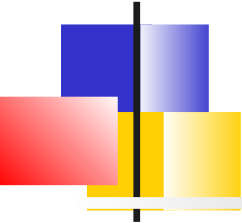
UserInfo endpoint

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 retournées peuvent être configurées

Il est accessible avec un jeton d'accès.

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



Authentication avec OpenID

Discovery endpoint
Authentication
Jeton d'identification et userinfo
endpoint
Personnalisation du jeton
Logout



Introduction

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

OAuth2 prévoit les scopes pour accéder à certaines informations sur le user

Keycloak lui propose 2 mécanismes supplémentaires intégrant les scopes OAuth2 :

- Les Protocol Mappers
- Les clients scopes



Protocol Mappers

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

- Rôles, revendications et attributs personnalisés codés en dur.
- Utiliser les métadonnées utilisateur pour remplir un jeton.
- Renommez les rôles.

Lors de la définition du mapper, on indique si l'information récupérée sera présente dans le jeton d'identification et/ou d'autorisation

Keycloak fournit des mappers prédéfinis.

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

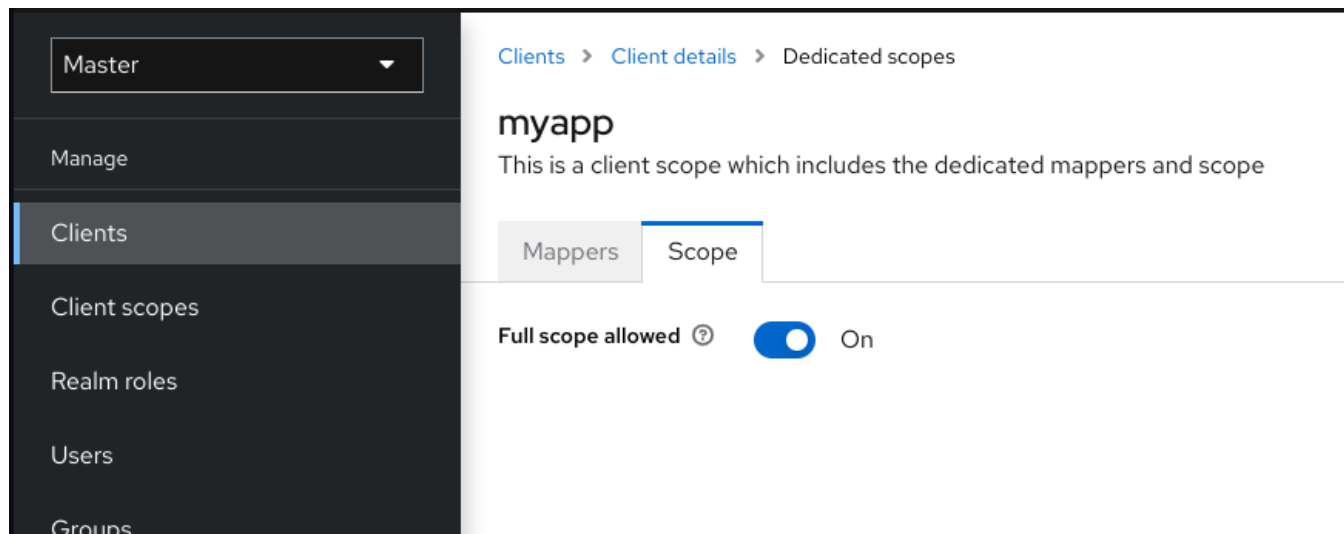


Role scope mapping

Role scope Mapping permet de limiter les rôles déclarés 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.

Par défaut, chaque client obtient tous les rôles de l'utilisateur.





Client Scopes

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

Cela englobe principalement

- Les mappers
- Les mapping de rôles auxquels le client a accès

Les *client scopes* prennent en charge le paramètre scope d'OAuth 2. permettant de demander des revendications ou des rôles en fonction des exigences de l'application.



Client Scopes

Keycloak propose un ensemble de 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)
- **microprofil-jwt** : Gère les revendications définies dans la spécification MicroProfile/JWT.
- **offline_access** : Spécification OpenID Connect.
- **profile, email, address, phone** : Défini dans OpenID Connect des mappers mais pas de rôle mapping



Liens entre Client et Client Scopes

La liaison entre un scope client et un client est configurée dans l'onglet Client Scopes du client

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 a été précisé dans la requête d'authentification d'OpenID



Evaluer 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

Lors de l'émission de jetons à un utilisateur, le client scope s'applique uniquement si l'utilisateur est autorisé à l'utiliser.

- Lorsqu'aucun rôle mapping n'est défini, chaque utilisateur est autorisé à utiliser cette étendue client.
- Lorsque des rôles mapping sont définis, l'utilisateur doit être membre d'au moins l'un des rôles.
Il doit y avoir une intersection entre les rôles d'utilisateur et les rôles du scope client.



Client scope dédié

Chaque client a un client-scope dédié qui est automatiquement appliqué quelque soit le scope oauth2 demandé.

On peut y définir

- Des mappers
- Des mapping de rôle



Authentication avec OpenID

Discovery endpoint
Authentication
Jeton d'identification et userinfo
endpoint
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 un action manuelle de l'utilisateur

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

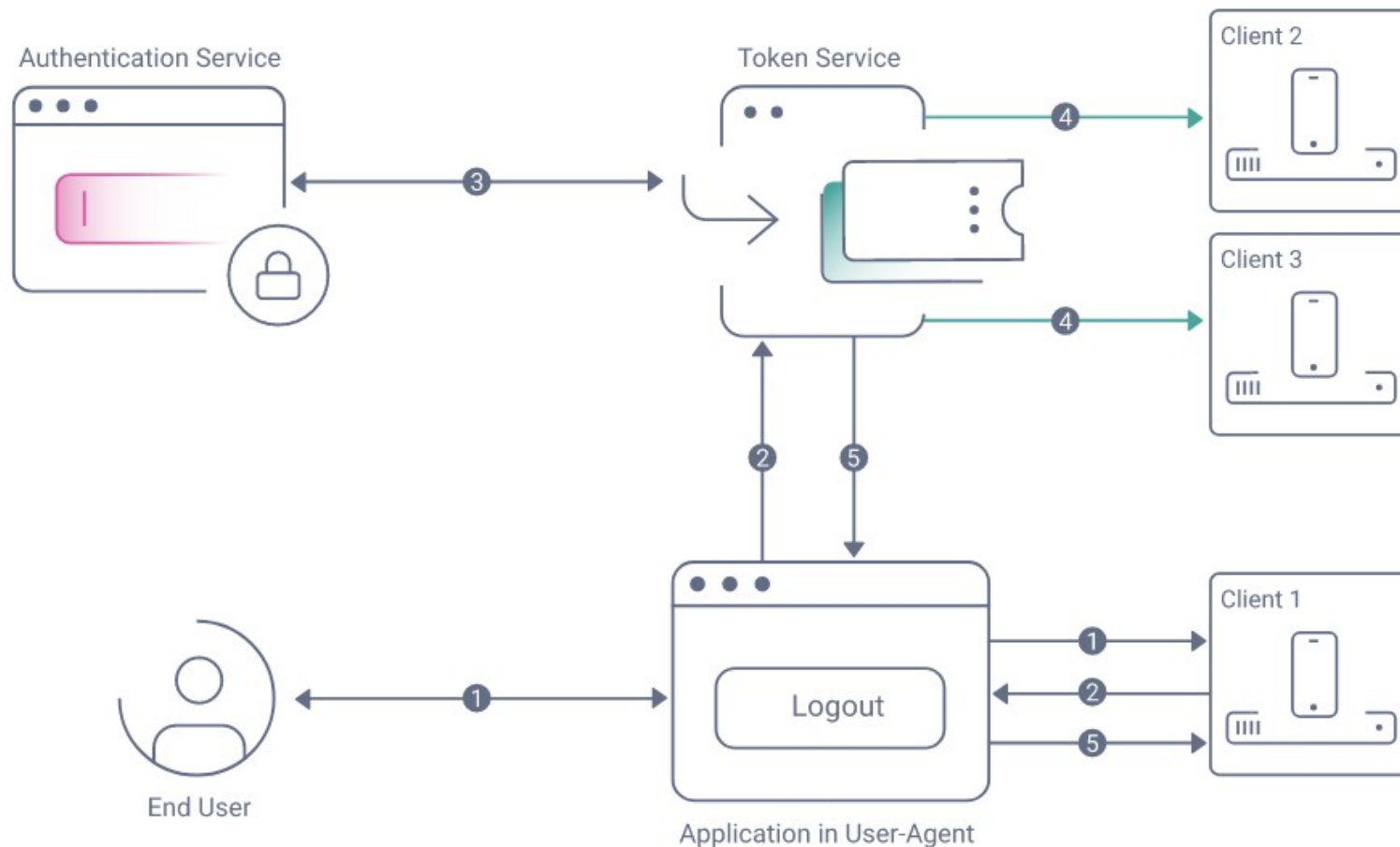
La déconnexion doit également provoquer l'invalidation des jetons actifs de l'utilisateur



Mécanisme du logout

1. Une déconnexion peut être initiée par l'utilisateur en cliquant sur un bouton de déconnexion dans l'application.
2. L'application nettoie son contexte de sécurité et redirige l'utilisateur vers le *end_session_endpoint* de Keycloak avec éventuellement les paramètres suivants :
 - ***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. Les jetons 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*

OpenID Connect Logout



Découverte de la déconnexion

On peut s'appuyer sur différents moyens, afin qu'une application s'aperçoive de la déconnexion :

- En s'appuyant sur la courte validité des jetons ID et accès, la déconnexion est détectée au moment du rafraîchissement du jeton ou de l'utilisation du jeton d'accès
- **OpenID Connect Session Management** : L'application obtient régulièrement des informations sur l'état de la session de Keycloak. (iframe effectuant un poll)
- **OIDC Back-Channel Logout** : Une application peut enregistrer un point d'accès pour recevoir les événements de déconnexion. A la déconnexion, Keycloak envoie un jeton de logout sur cette URL
- **OIDC Front-Channel Logout** : Enregistrement d'une URI de logout. La page de logout de Keycloak contient des *iframes* avec les URI de logout de chaque client.
Complicé et peut être bloqué par CSP (Content Security Policies)



Autoriser les accès avec oAuth2

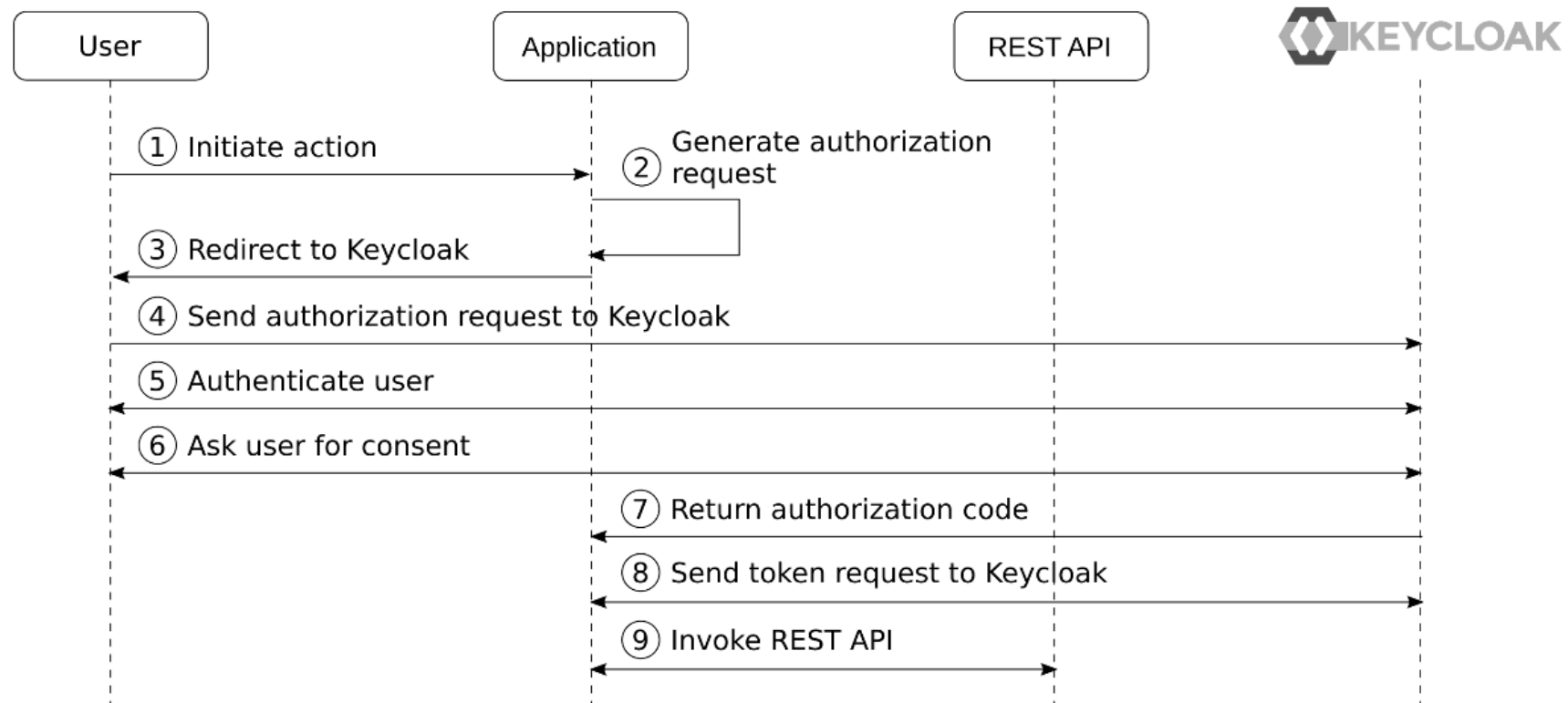
Jeton d'accès et consentement

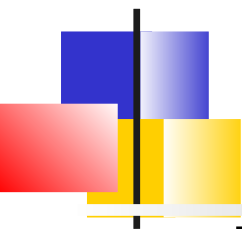
Limitations des accès

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 demandées par l'application.

- Les scopes peuvent être demandées *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 est de confiance et l'administrateur qui a enregistré l'application auprès de Keycloak peut pré-approuver l'accès au nom de l'utilisateur
- C'est la valeur par défaut de Keycloak 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.

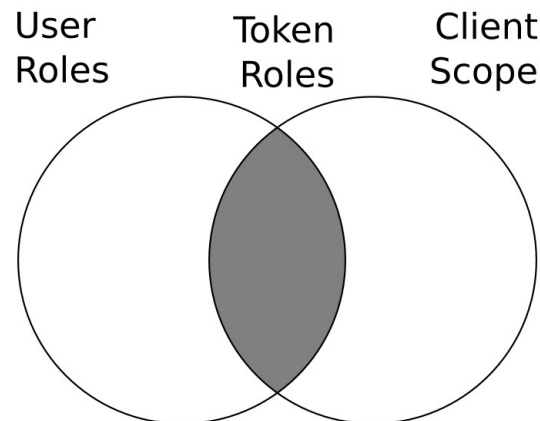
- Manuellement en ajoutant un client spécifique à l'audience à l'aide d'un mapper.
On peut par exemple utiliser le scope client dédié au client toujours présent dans le jeton
- Automatiquement, si le client a une scope sur un rôle client d'un autre client.

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 le 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 reste l'*Authorization Code Flow*

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



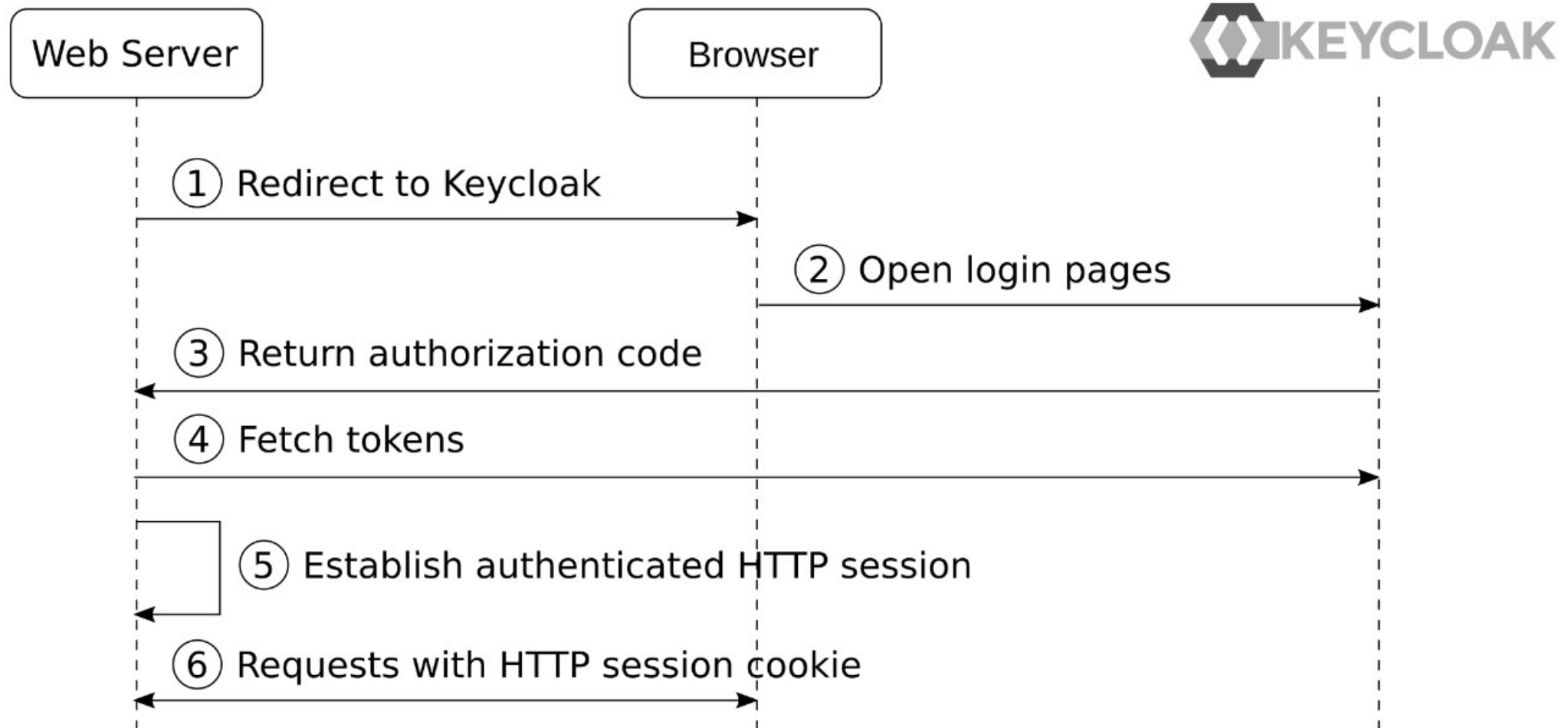
Application traditionnelle

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 :

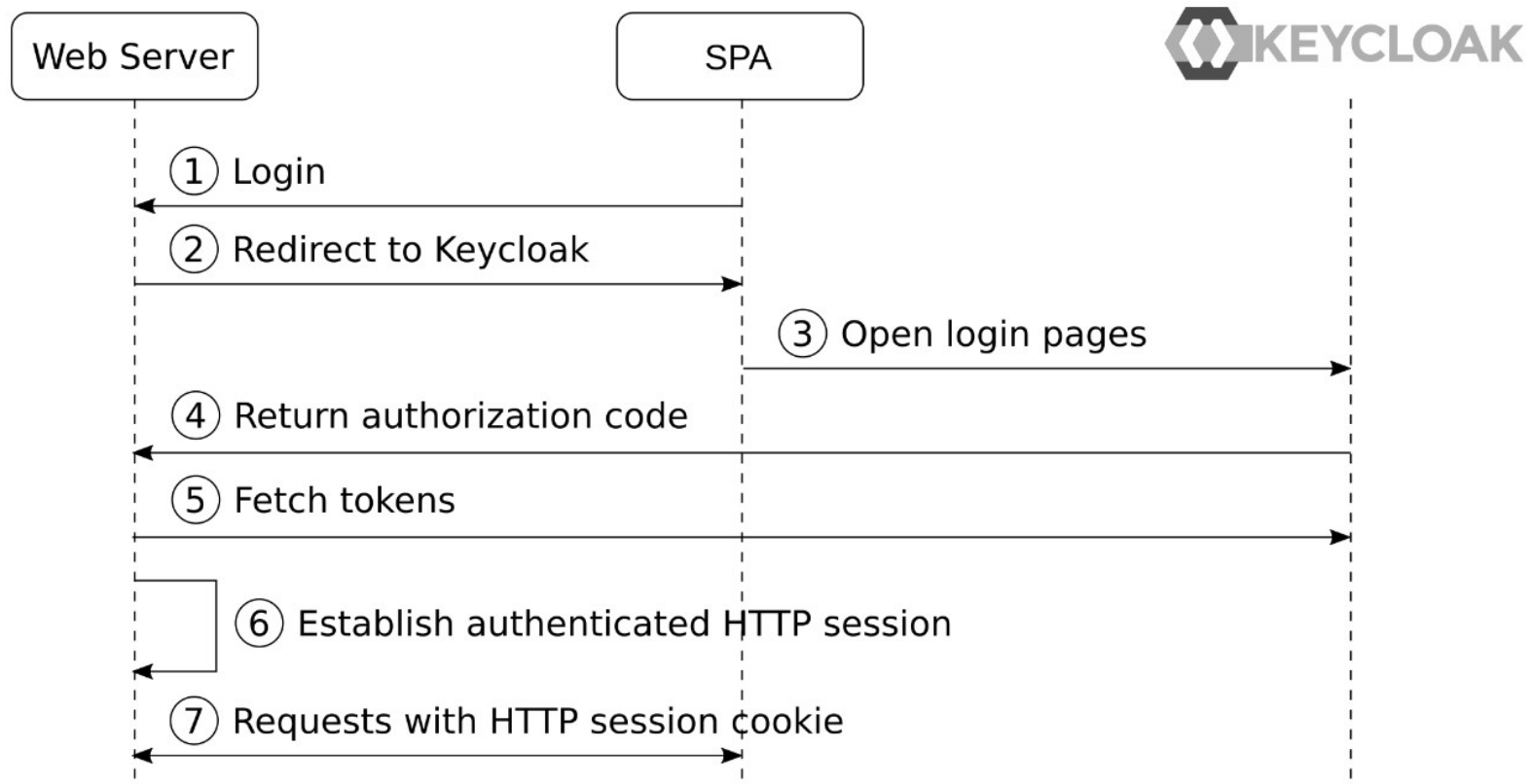
- Enregistrer un client confidentiel
Pour obtenir le jeton, il faut alors le code + le secret du client stocké dans le back-end
- On peut également configurer *PKCE* qui protège contre d'autre type d'attaque
- Donner des URLs de redirection strictes

Application traditionnelle



SPA avec API dédiée

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





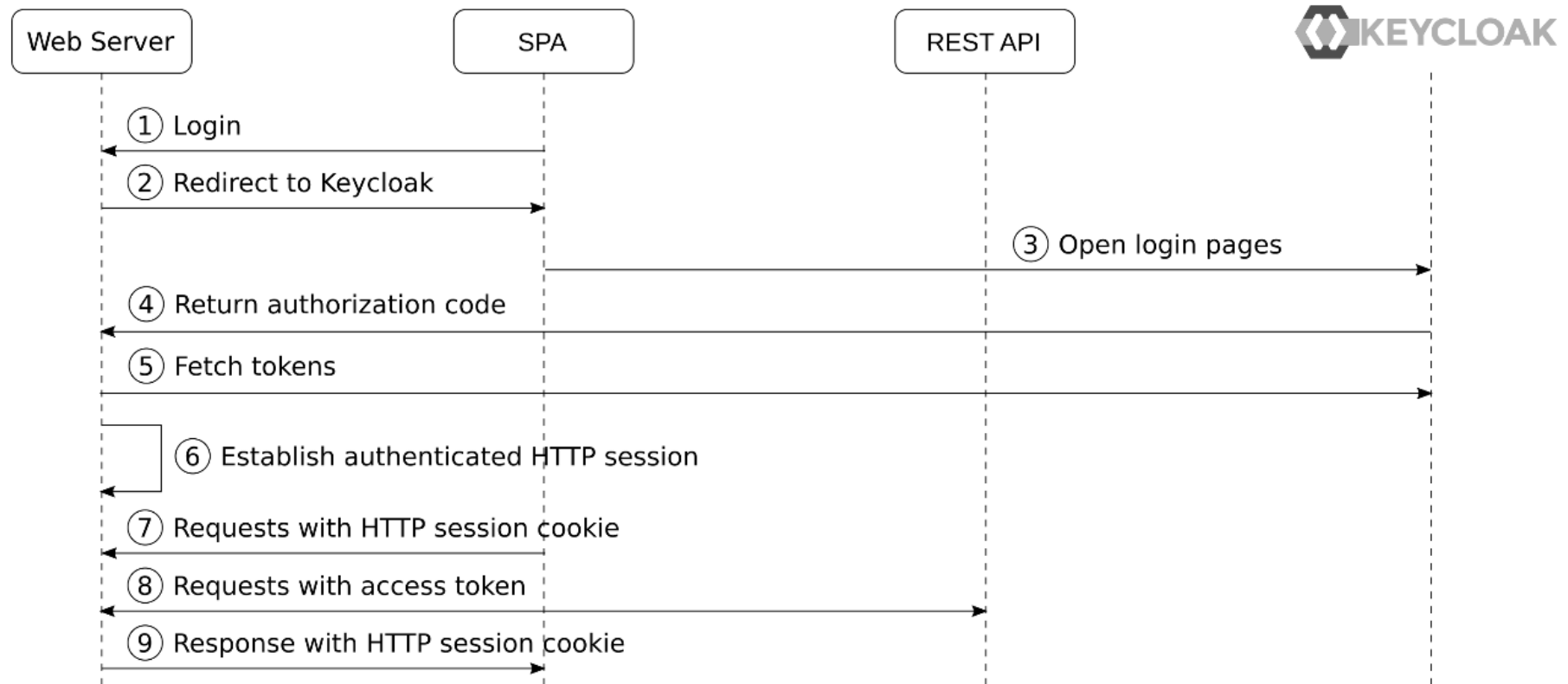
SPA avec API intermédiaire dédiée

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





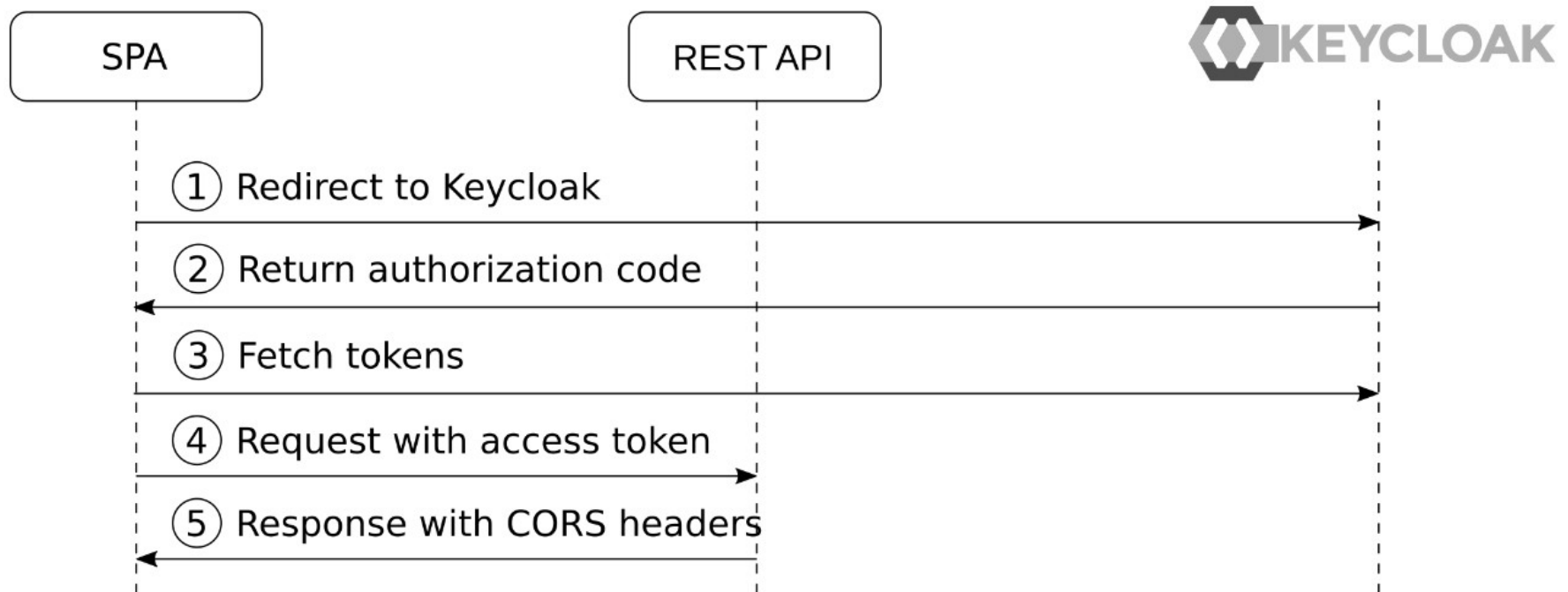
SPA avec API externe

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

=> *Authorization Code flow* avec l'extension PKCE

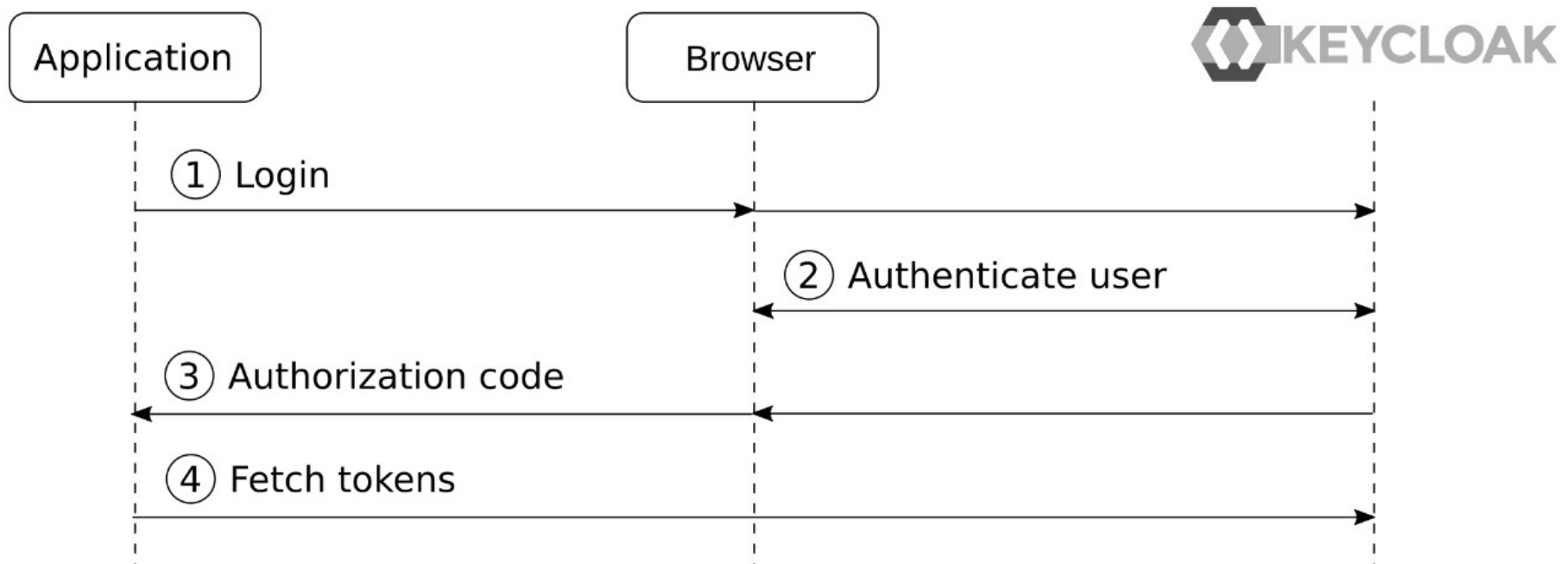


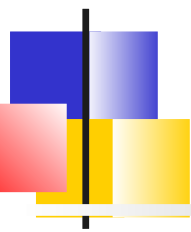
Authentication

L'application native ou mobile doit donc utiliser un navigateur pour l'authentification :

- Utilisez une vue Web intégrée dans l'application
Inconvénients : Vulnérabilité à l'interception de crédentiel, plus de SSO (pas de partage de cookies entre plusieurs applications)
- Utilisez des fonctionnalités de certaines plateformes comme Android et iOS : in-app browsers.
Permet de profiter du navigateur système et de rester dans l'application
=> Recommandé
Cependant, une application malicieuse pourrait utiliser cette technique pour collecter les crédentiels en proposant une fausse page de login
- Utilisez un agent utilisateur externe : le navigateur par défaut de l'utilisateur.
Inconvénient : On sort de l'application

Flow





Renvoi du code d'autorisation

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

- **Claimed HTTPS scheme** : Certaines plates-formes (Android, iOS) permettent à une application de revendiquer un schéma HTTPS. 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
- **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
- **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.

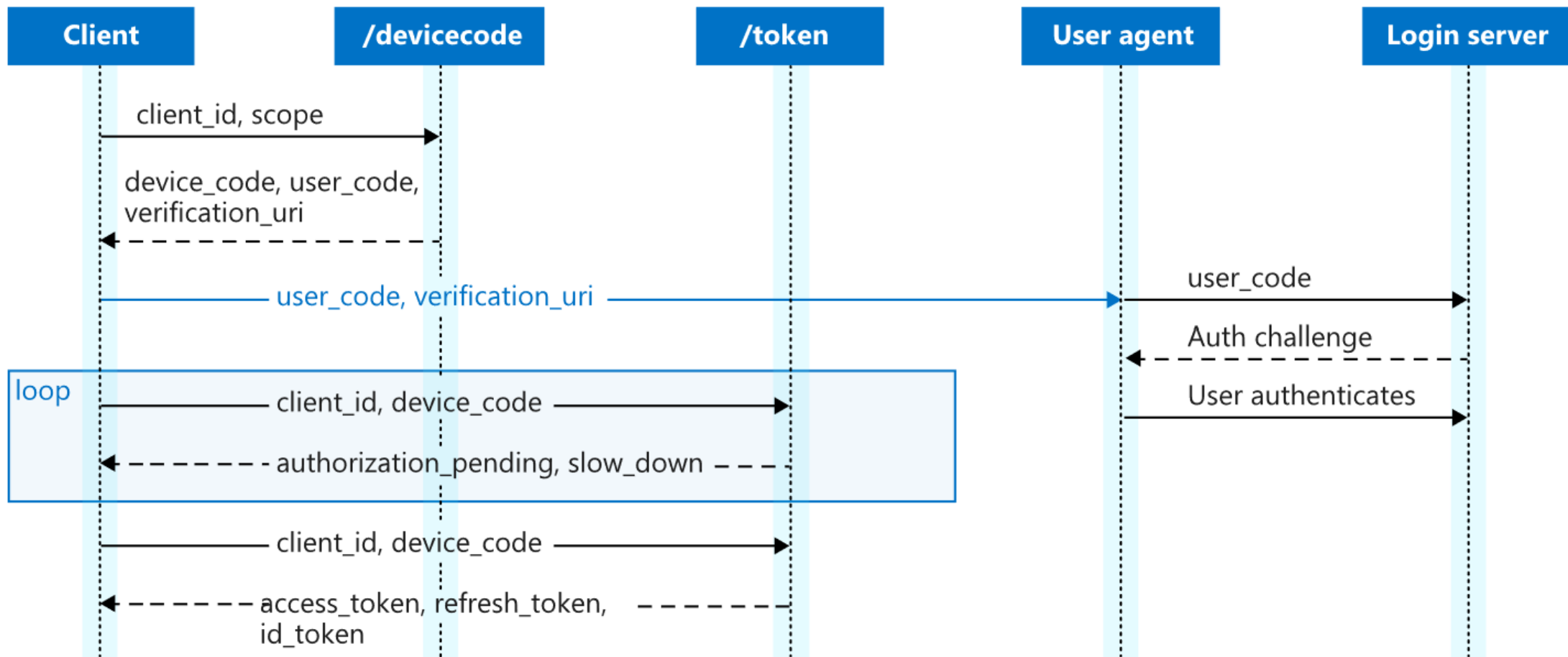


Device Code Flow

Lorsqu'un navigateur n'est pas disponible, il faut se tourner vers le **Device Code flow**

- 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 code device

Device Code Flow





Sécurisation des différents types d'application

Application web
Application native ou mobile
REST APIs et services



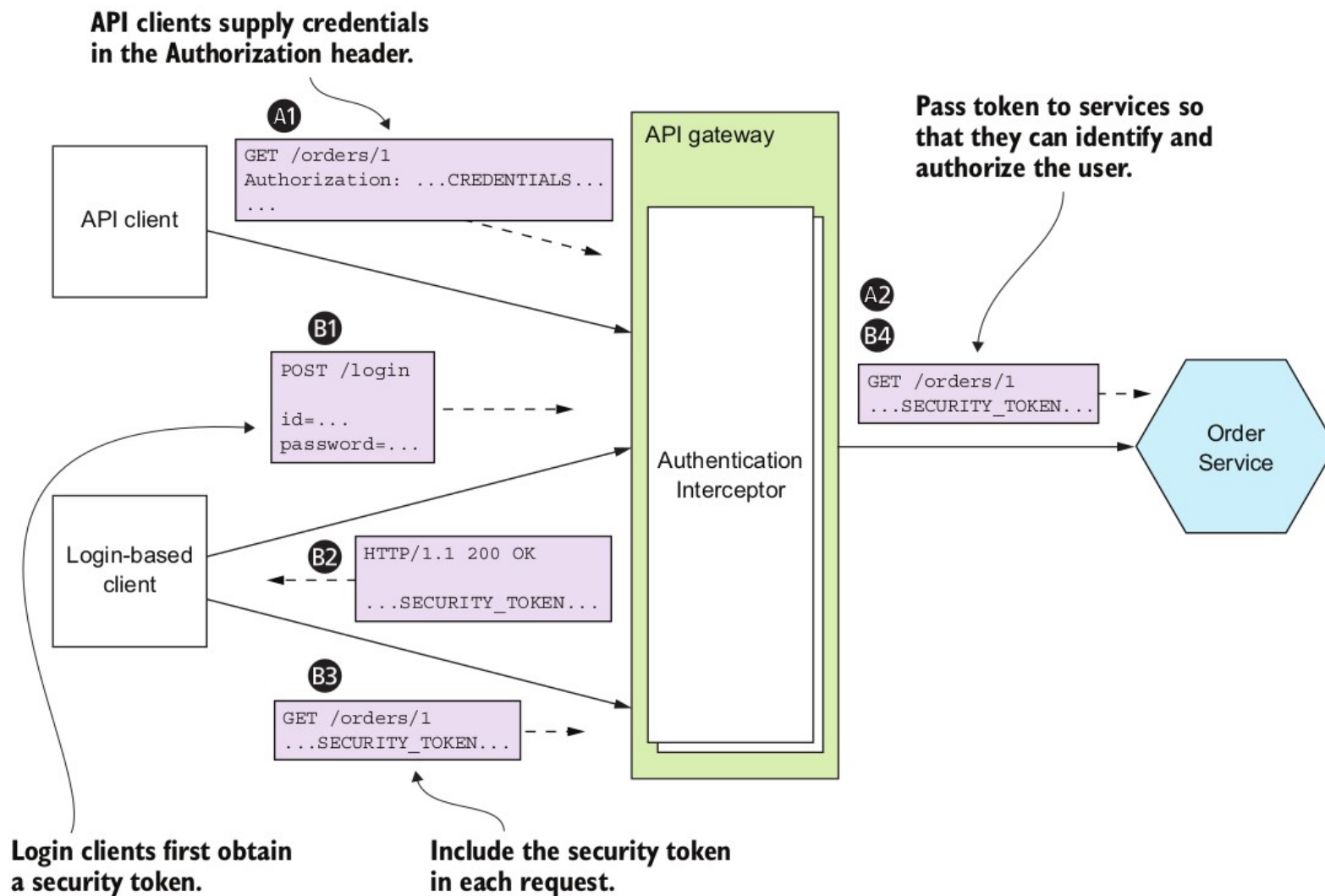
Introduction

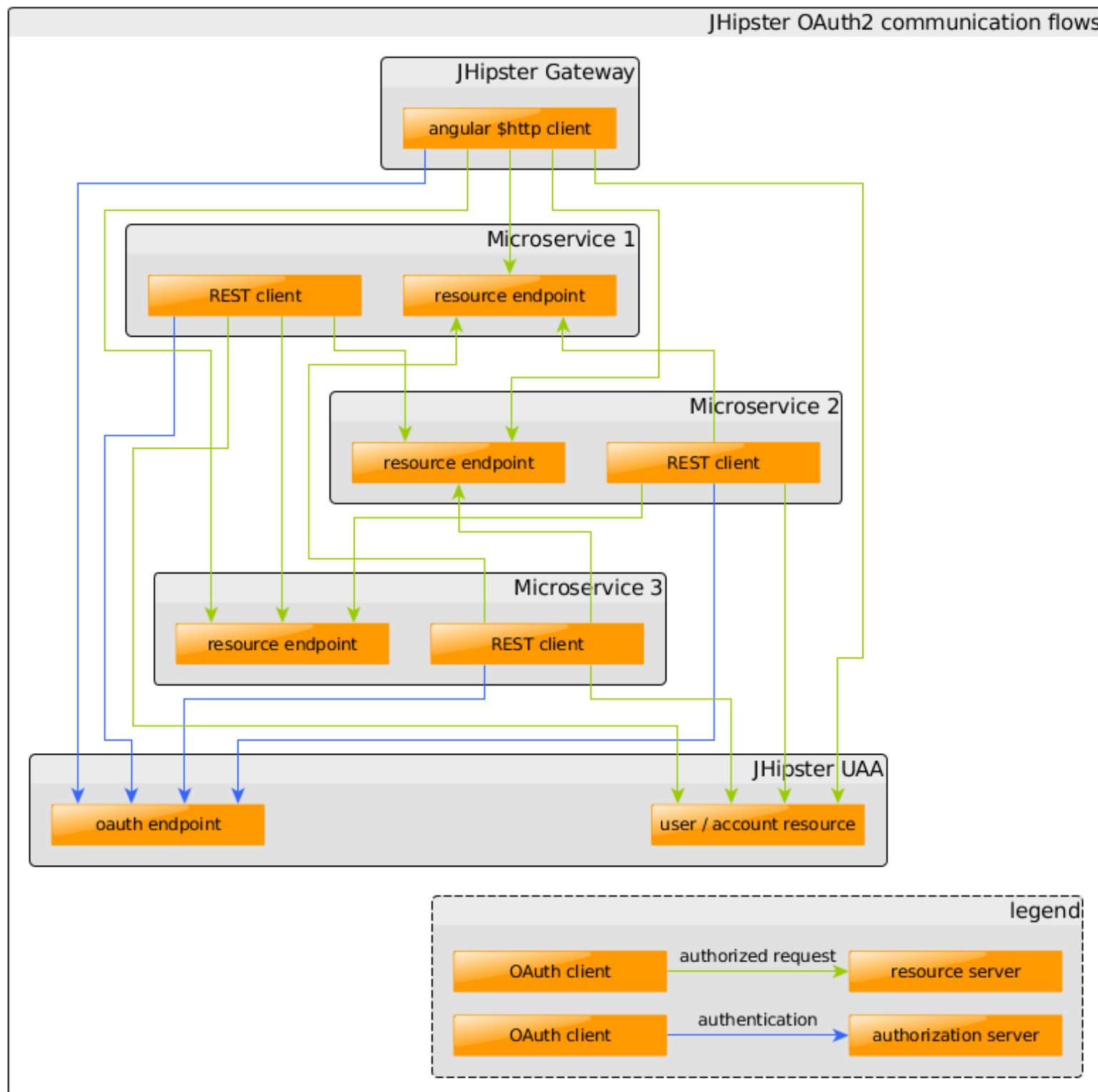
Lorsqu'une application souhaite invoquer une API protégée par OAuth2, elle obtient d'abord un jeton d'accès de Keycloak, puis inclut le jeton d'accès dans l'en-tête d'autorisation.

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

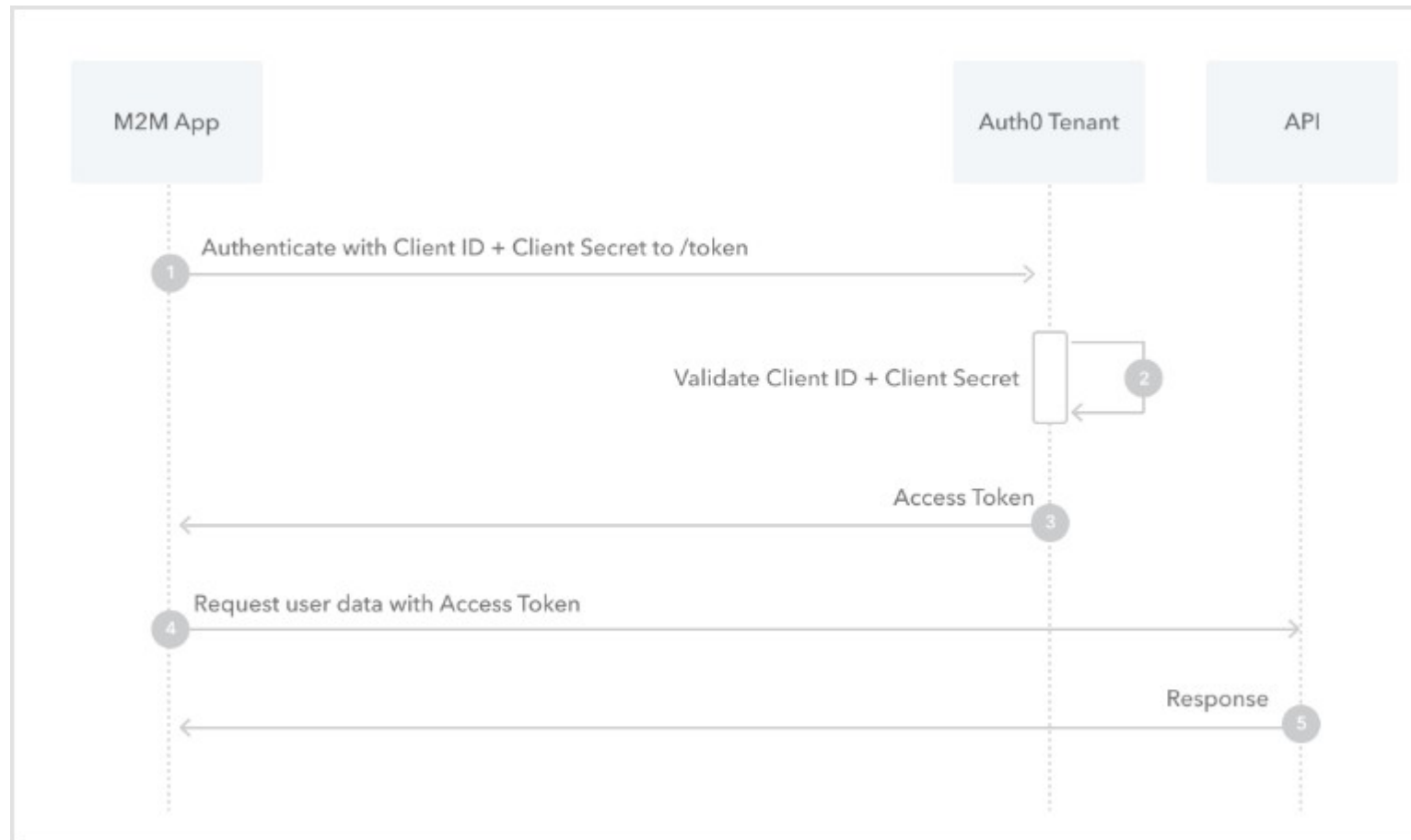
- Propagation de jeton, le même jeton est propagé lors des appels inter-services. Tous les micro-services partagent le même contexte de sécurité
- Chaque micro-service utilise son propre jeton obtenu avec un *Client Credentials grant*. Chaque interaction a alors son propre contexte de sécurité

Access Token Pattern





Client Credentials Flow





Intégration KeyCloak

Introduction

Adaptateurs Keycloak

SpringBoot

Quarkus

Reverse Proxy

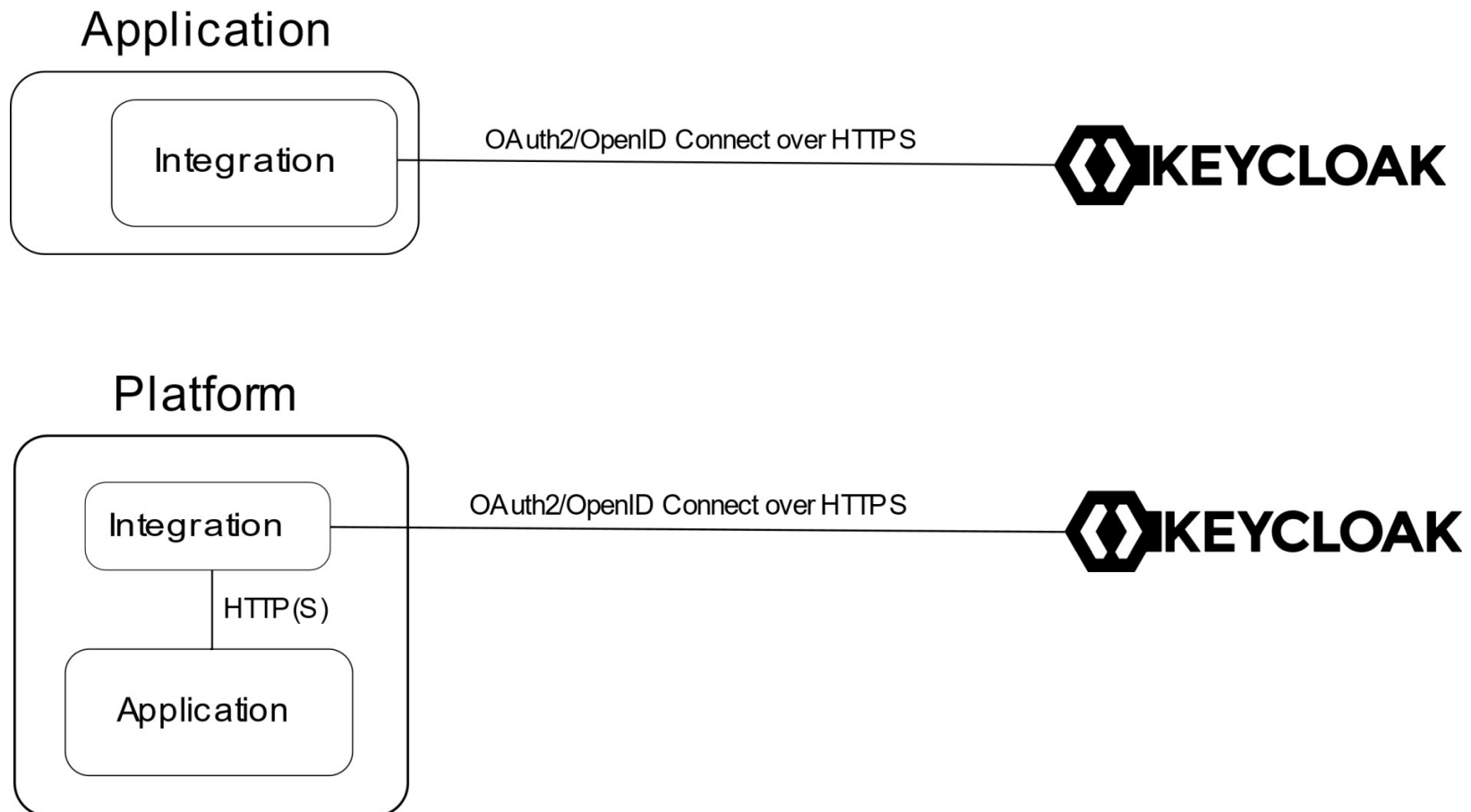


Introduction

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 : Utilise 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 dans différents langages

- 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/**](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 Keycloak.

Seul l'adaptateur Javascript sera maintenu.

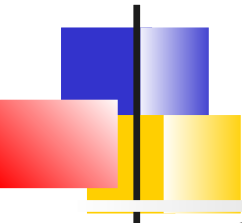


Adaptateurs Keycloak

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

Instanciación : **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
- **checkLoginframe** : 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
- ***Okta*** : Pour travailler avec le fournisseur *oAuth* Okta



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());

    // ...
}
```

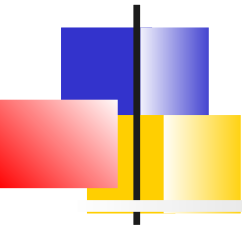


Logout

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("/"))
```

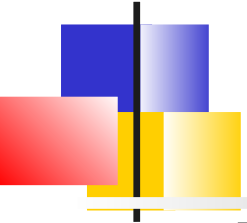



LogoutHandler

```
private void logoutFromKeycloak(OpenIdUser user) {
    String endSessionEndpoint = user.getIssuer() + "/protocol/openid-connect/logout";

    UriComponentsBuilder builder = UriComponentsBuilder
        .fromUriString(endSessionEndpoint)
        .queryParams("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/realms/<realm-name>
          jwk-set-uri: http://keycloak:8083/realms/<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(jwtGrantedAuthoritiesConverter);  
  
    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 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

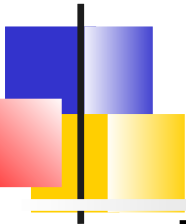
```
quarkus.oidc-client.auth-server-url=  
    http://localhost:8180/auth/realms/quarkus
```

Découverte manuelle

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

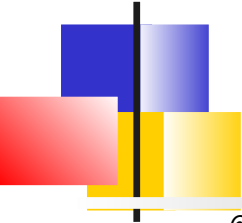
L'extension ***quarkus-oidc-token-propagation*** fournit 2 implémentations de *ClientRequestFilter* qui simplifient la propagation des informations d'authentification

- ***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/realms/<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 inclut dans les jetons. Cela nécessite un mapper 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 protègent 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

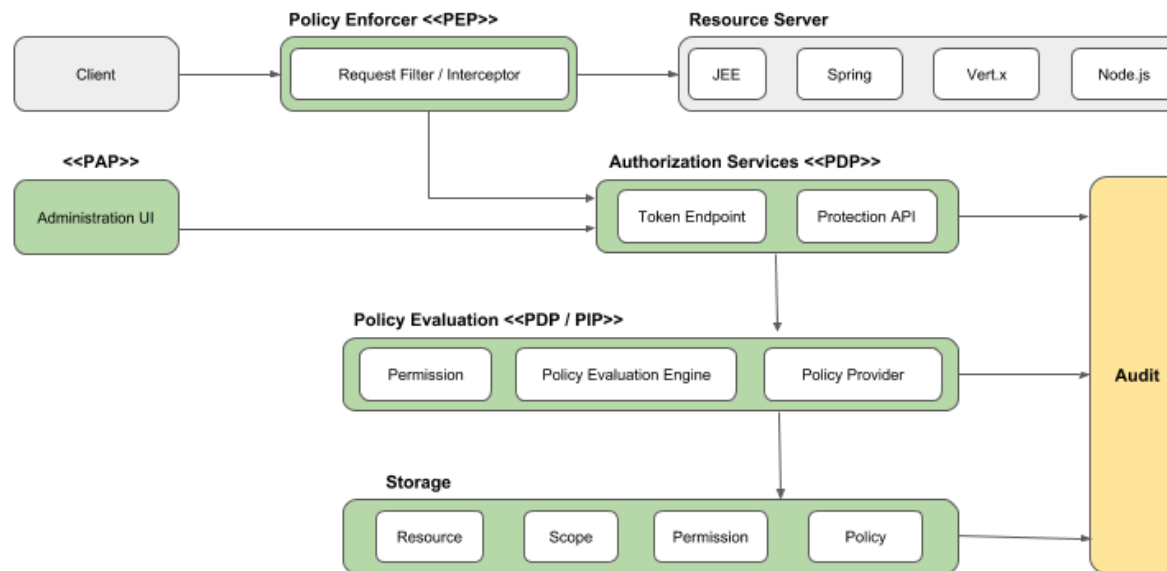


Fonctionnalités

Keycloak autorisation services est capable de combiner différents mécanismes 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 \Leftrightarrow Client
Ressources \Leftrightarrow Motif d'URI
Scopes (Optionnel) \Leftrightarrow 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.

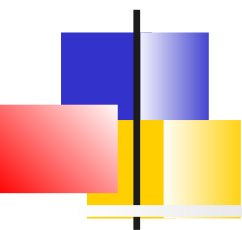
Les autorisations sont appliquées en fonction du protocole que l'on utilise :

- 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.kantarinitiative.org/uma/wg/rec-oauth-uma-grant-2.0.html>

2. Requesting Party Token





Administration Keycloak

Gestion des utilisateurs

Authentification des utilisateurs

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 crédits
- 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**

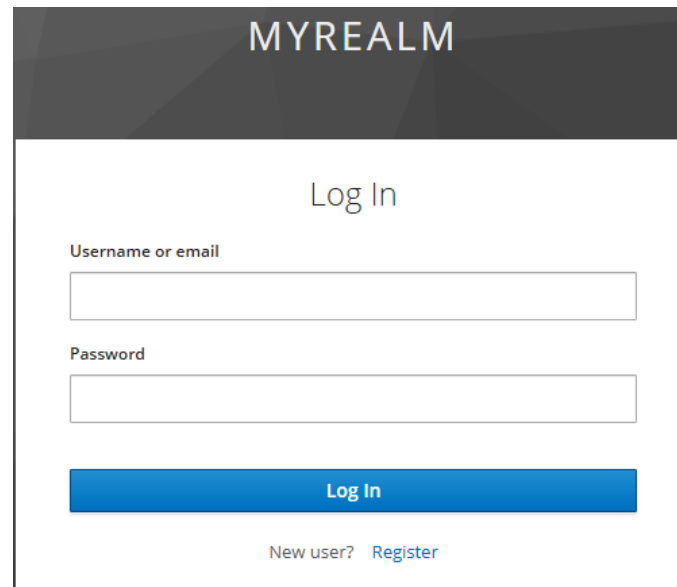
- **Verify Email** : envoyer un e-mail à l'utilisateur pour confirmer qu'il appartient à cet utilisateur.
- **Update Password**: demandez à l'utilisateur de mettre à jour son mot de passe.
- **Update Profile**: demandez à l'utilisateur de mettre à jour son profil en fournissant son prénom, son nom et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



Self-registration

Keycloak permet aux utilisateurs de s'enregistrer par eux même

Realm Settings → Login → User registration



MYREALM

Log In

Username or email

Password

Log In

New user? [Register](#)



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



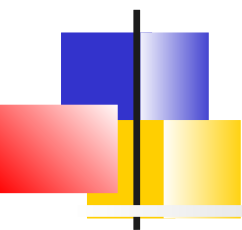
Autres fournisseurs d'identité

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





Administration Keycloak

Gestion des utilisateurs

Authentication des utilisateurs

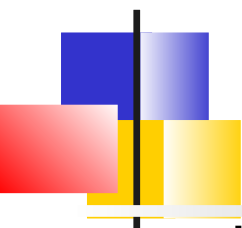
Gestion des sessions et jetons



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



OTP

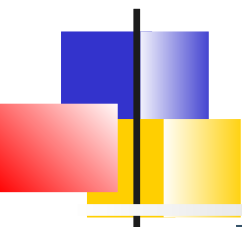
En plus de fournir un mot de passe, les utilisateurs doivent fournir une 2ème preuve de leur identité

- 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

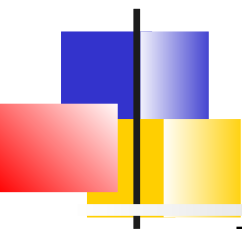
Authentication → Policies → OTP Policy



WebAuthn

WebAuthn est basé sur des clés asymétriques - une paire de clés privée-publique - pour enregistrer des appareils des utilisateurs et les authentifier dans un système.

- Il permet d'utiliser ces appareils pour 2FA, MFA, ou pour authentifier de manière transparente les utilisateurs sans aucun mot de passe
- WebAuthn est plus sécurisé que OTP car il n'y a pas de clé partagée entre Keycloak et les applications tierces (FreeOtp et GoogleAuthenticator)
- Le device doit être conforme aux exigences de FIDO2. Smartphone prenant en charge les empreintes digitales, clé de sécurité connectée via USB



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

Example Browser Flow

Master

Manage

Clients

Client scopes

Realm roles

Users

Groups

Sessions

Events

Configure

Realm settings

Authentication

Identity providers

User federation

Authentication > Flow details

Browser Default Built-in

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

+ Add step

+ 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 authentificateur 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. Onligatoire (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.



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



Mise en place Webauthn

La première étape consiste à configurer
comme action requise :

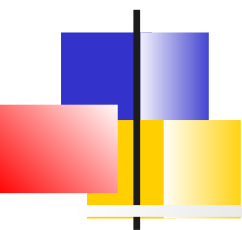
Webauthn Register

Puis définir un flow avec

WebAuthn authentication

Enfin configurer le client pour utiliser le
flow

Clients → <Client> → Advanced



Administration Keycloak

Gestion des utilisateurs

Authentification des utilisateurs

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** sont elles dédiées à l'authentification de l'utilisateur pour un client donné
Elles sont strictement liées à la validité des jetons et à leur utilisation par les applications



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



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

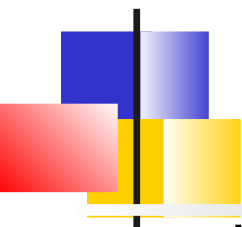


Cookies

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 la flag
secure est également positionné 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 suit la configuration des timeout idle des sessions SSO ou sessions clientes.



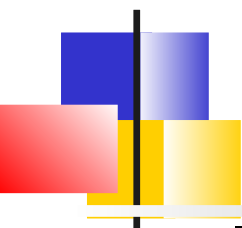
Configuration des jetons

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***



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/realms/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/realms/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



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
```



Éléments 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) de 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, Exécuter 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

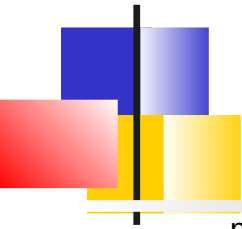


Interface factory

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



Example

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";
    }

```



Example

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.myboite.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

Comptes utilisateur et applications



Sources de configuration

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

- 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

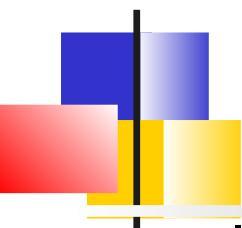
HTTPS/TLS requis



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 clé 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

Comptes utilisateur et applications



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 certificat, Keycloak s'appuie sur des fichiers PEM ou des keystore
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

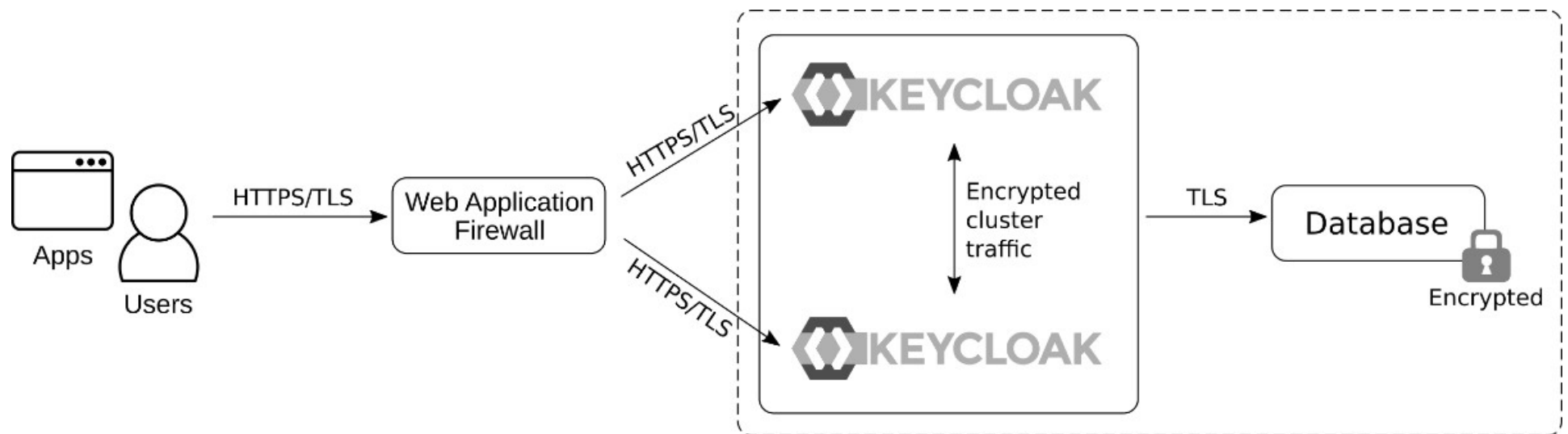
Configuration et optimisation du
démarrage

Configuration de production

Sécurisation Keycloak

Comptes utilisateur et applications

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 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

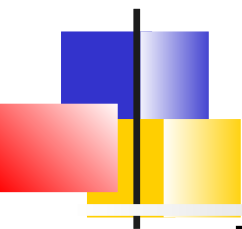


Sécuriser la BD

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



Vers la production

Configuration et optimisation du
démarrage

Configuration de production

Sécurisation Keycloak

Comptes utilisateur et applications



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

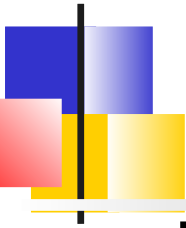


Sécurité des applications Web

Comprendre les vulnérabilités des applications Web en consultant OWASP¹

Appliquer ses recommandations :

- Authentification : S'assurer que les sessions sont sécurisées
- Autorisation : Donner toujours le minimum de privilèges
- Comprendre les TOP10 de OWASP
- Mettre à jour régulièrement les librairies et les frameworks utilisés
- Crypter les données sensible
- Trace et surveillance afin de détecter les attaques
- Firewall



Recommendations OAuth2.0

Les spécifications OAuth2 et OpenID Connect sont très flexibles et certaines configurations ne sont pas sécurisées

Voir <https://oauth.net/2/>

- OAuth 2.0 for mobile and native apps
- OAuth 2.0 for browser-based apps
- OAuth 2.0 threat model and security considerations
- OAuth 2.0 security best current practice

OAuth 2.1 intègre plusieurs bonnes pratiques dans la spécification.



Financial-Grade API (FAPI)

FAPI est un groupe de travail qui essaie de spécifier des profils de configuration OIDC pour le domaine bancaire. 2 profils sont définis :

- FAPI 1.0 – Part 1: Baseline API Security Profile

https://openid.net/specs/openid-financial-api-part-1-1_0.html

- FAPI 1.0 – Part 2: Advanced Security Profile

https://openid.net/specs/openid-financial-api-part-2-1_0.html

Keycloak 15.0.2 a été certifié FAPI OpenID Provider en janvier 2022



Configuration des applications clientes

Consentement requis : Activé cette option pour toute application tierce.

Type d'accès : Confidentiel si l'on peut conserver le crédentiel en toute sécurité côté serveur.

Implicit Flow Enabled : Jamais sauf pour application legacy

Direct Access Grants Flow Enabled: Jamais sauf pour application legacy

Valid Redirect URIs: Exact match



Algorithmes de signature

Keycloak supporte plusieurs algorithmes de signature :

- **Signature Rivest-Shamir-Adleman (RSA)** : Algorithme par défaut utilisé par Keycloak. Pas la plus sécurisée, mais la plus largement disponible.
- **Elliptic Curve Digital Signature Algorithm (ECDSA)** : Plus sûr que RSA et est également beaucoup plus rapide. Recommandé si possible
- **Hash-based message authentication code (HMAC)** : Algorithme de signature symétrique qui nécessite l'accès à un secret partagé.

On peut également choisir entre différentes longueurs de hachage de signature.

Avec des jetons à durée de vie relativement courte, une longueur de 256 bits est considérée correcte