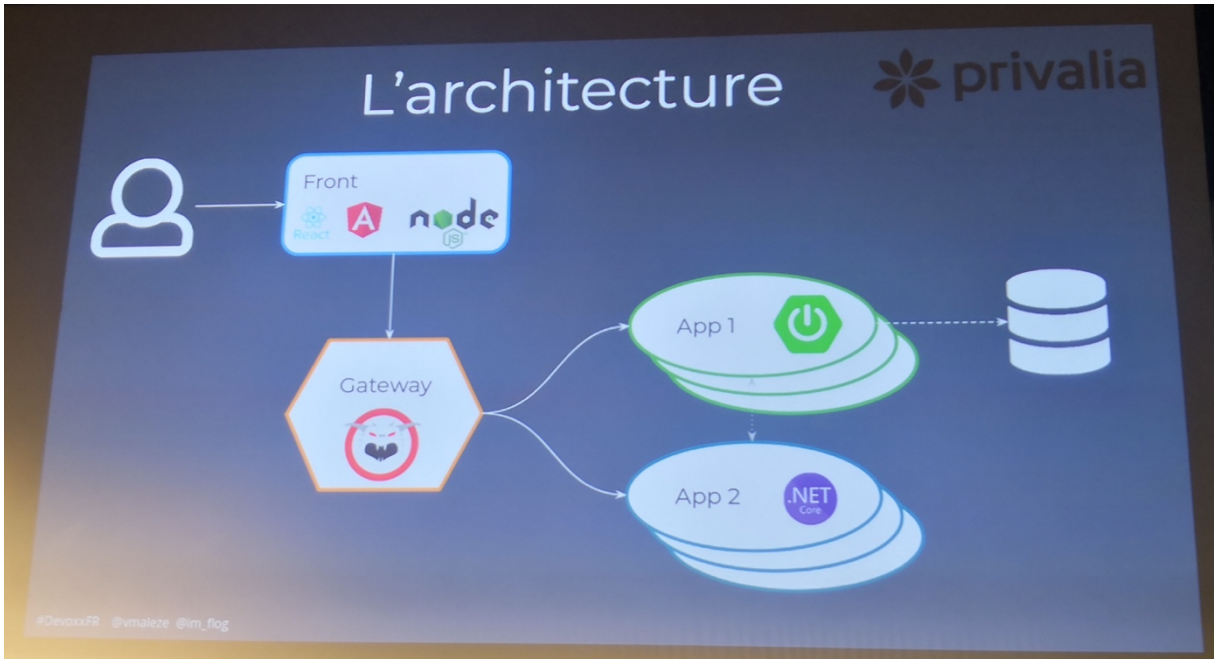
La gestion de l'authentification et de l'autorisation dans une architecture microservices ? Pas de soucis !

Speakers : Vivien Maleze (Ippon Technologies), Florian Garcia (Betclic)
Format : Conférence
Date : 18 avril 2019
Slides et code : https://github.com/ImFlog/microservices-security

Les thèmes abordés au cours de cette université sont : authentification, autorisation, OAuth 2, JWT, sessions

Cas d'une architecture inspirée de Vente Privée :

- Front en NodeJS
- Gateway
- Applications Spring Boot et .NET Core



Dans cette architecture, il n'y avait pas d'authentification (on ne dit pas qui on est) ni d'autorisations (droits). Bien qu'on soit dans un réseau privé, n'importe quel développeur pouvait appeler n'importe quel WS de back-office. On a besoin d'ajouter un SSO.

Besoins en termes de serveur d'identité :

- SSO : lorsqu'on se connecte à différentes applications, on a un unique point d'entrée (une seule mire de login)
- Fédération d'identité
- Social login : se logger avec GitHub, Instagram …
- Gestion des rôles
- Facilité d'administration
- Intégration avec Spring, Dot.NET, Nodejs
- Coûts (maintenance / Saas)

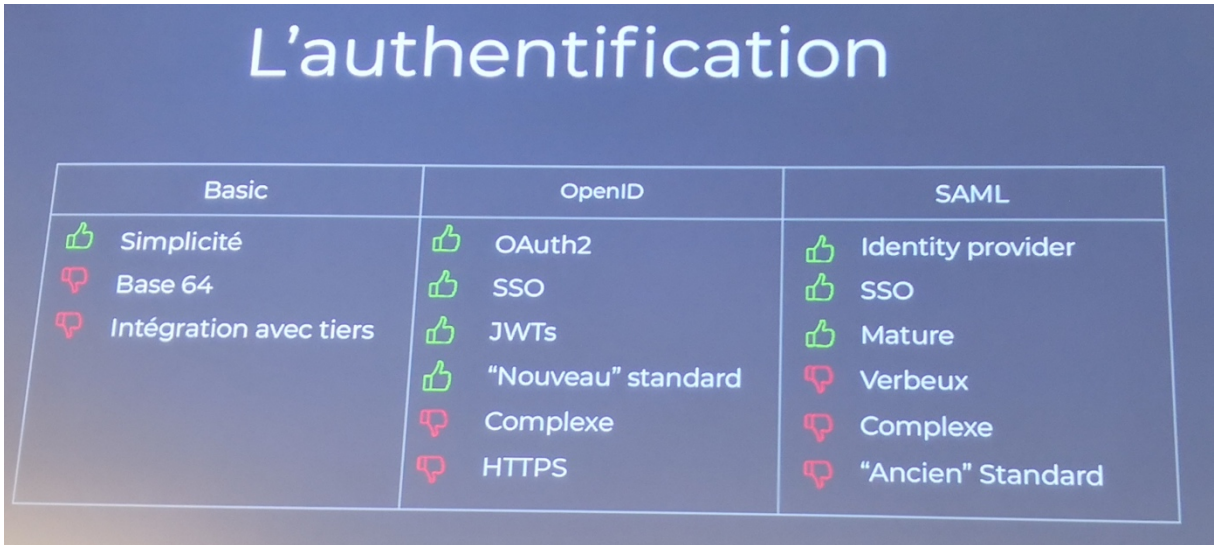
Une étude des différents acteurs du marché entre : okta, Auth0, Keycloak et implémentation maison. Faire de la sécu à la main pour une question aussi importante que la sécurité a vite été écartée. Les 3 autres solutions pouvaient convenir. Mais Vente Privée avait déjà un **Keycloak**. Ils sont partis dessus.

Keycloak

- Une seule mire de login (SSO)
- Supporte plusieurs protocoles : OpenID, SAML, Social Login
- Fédération d'identité : LDAP, Active Directory

Mécanismes d'authentification

- Basic HTTP : simple mais pas secure
- SAML : un peu ancien
- OpenID : norme basée sur OAuth 2 avec des jetons JWT



OAuth 2 est un protocole standardisé. Le mécanisme le plus courant est celui de « l'autorisation code ».
Dans le monde OAuth 2, le client est une application (à ne pas confondre avec l'utilisateur).
Le jeton (token) renvoyé par Keycloak peut être utilisé pour appeler des API.

Démo avec Keycloak et Postman pour montrer les requêtes OAuth 2 en utilisant un compte Google.
Utilisation de https://jwt.io pour décodé le token JWT.
Dans le Payload, on retrouve des informations standards OpenID et d'autres infos de Google.
Recommandation : ne pas mettre d'informations sensibles dans un token.

Lors de l'appel à Keycloak, on a reçu 3 tokens :

1. Id token
2. Access token :
3. Refresh token : utilisé pour rappeler le serveur d'identité

Tous ces tokens sont signés
Le endpoint/token/instrospect de Keycloak permet de valider le token. On peut le vérifier soit même en utilisant la clé publique mise à disposition par Keycloak.

Côté front (Anguler + NodeJS), nécessité de stocker le token :

- Local storage
- Cookie : à privilégier

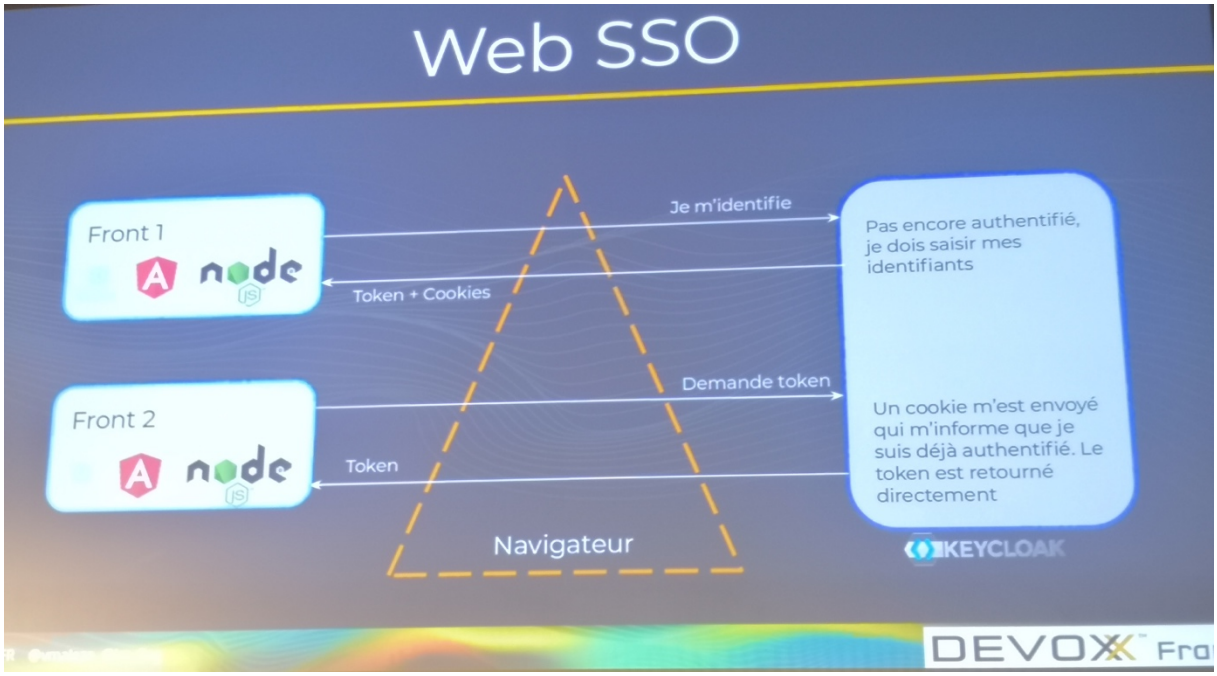
Le back est stateless et ne stocke donc pas le token.

Durée de vie des sessions :

- Short living token -> durée à régler par soit même (ex : 30 mn)
- Long living refresh token

Lorsque la durée de vie de l'access token a expiré, le client JS rappelle automatiquement Keycloak avec le refresh token.

Web SSO
Keycloak renvoie un cookie + image



Exemple de code d'un front Angular

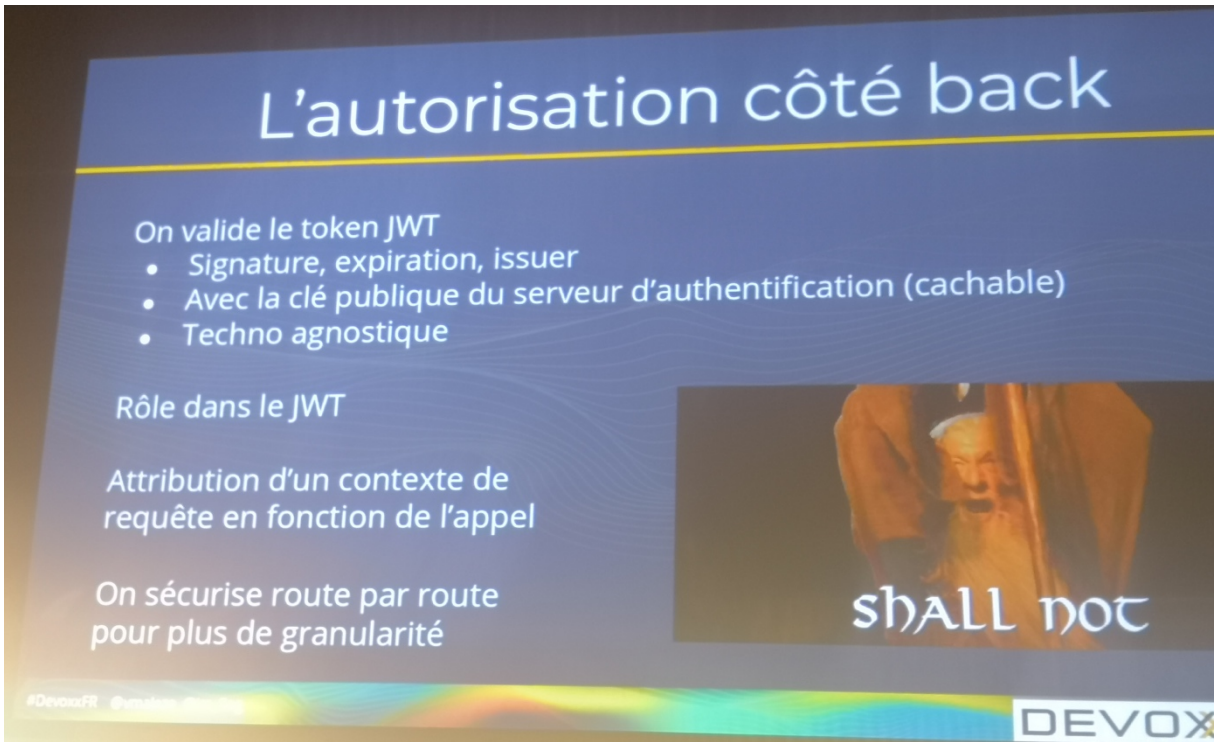
- Utilisation d'une guarde KeycloakAuthGuard à étendre. Lorsqu'on reçoit une requête, on vérifie que les rôles décrits dans le router correspondent biien au rôle de l'utilisateur authentifié

Comment gérer les utilisateurs ?

Dans Keycloak, il y'a une notion de royaumes permettant de gérer plusieurs clients.
On a 3 notions : Utilisateurs / Groupes / Rôles. Les rôles vont pouvoir être associés à des

utilisateurs et des groupes. Ils peuvent être peuplés par les LDAP / IDP /Services externes.

Comment valider le token côté back ?

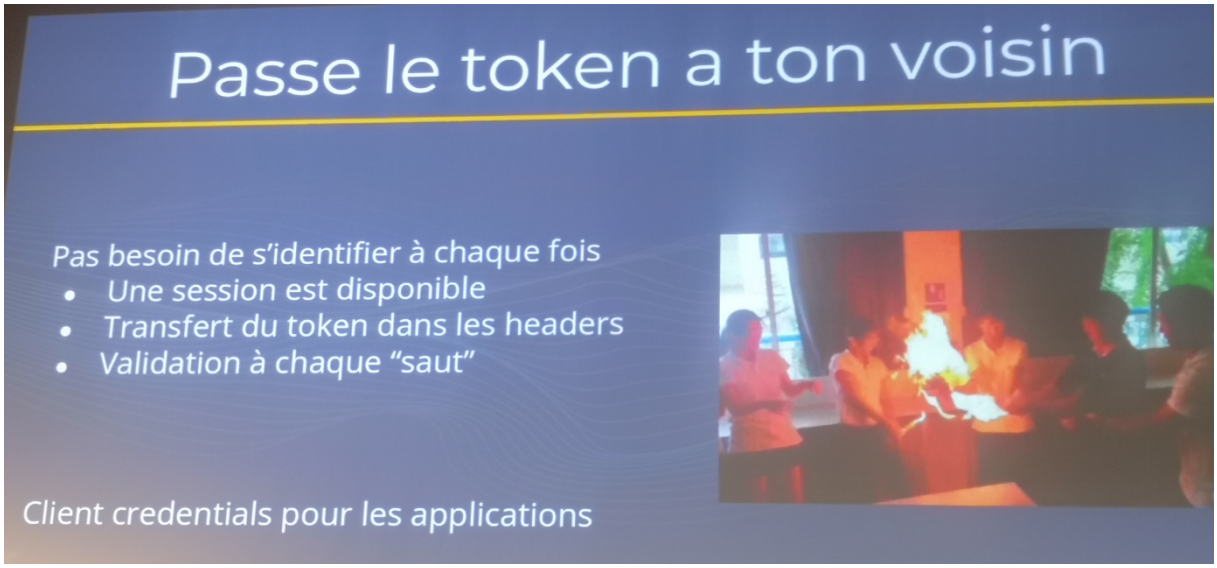


On valide le token JWT :

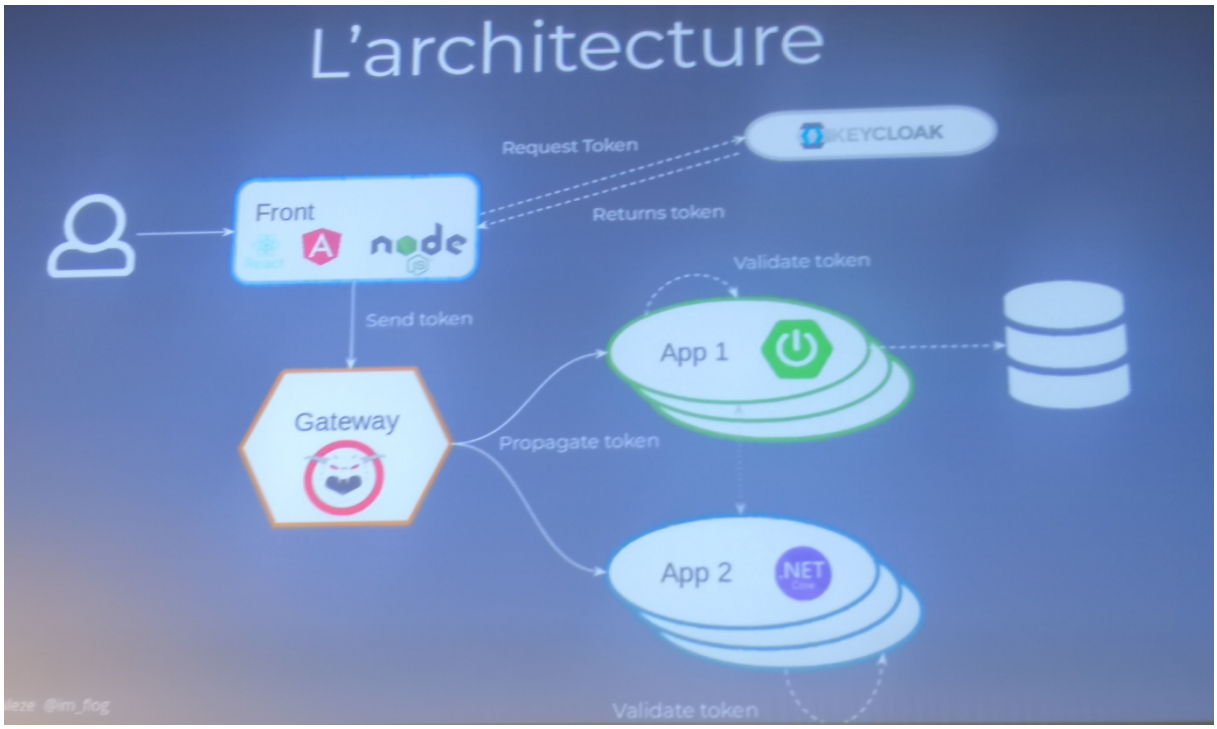
- Signature, expiration, issuer
- Avec la clé publique du serveur d'authentification (possobilité mise en cache)
- Techno agnostique

Le rôle de l'utilisateur est contenu dans le jeton JWT.

On peut passer le token à son voisin. Néanmoins, côté back, on vérifie le jeton à chaque fois.



Lorsque 2 applications souhaitent communiquer ensemble sans utilisateurs, on peut utiliser le « **Client credentials** » de Keycloak. Cela reste de l'OAuth 2.



Exemple côté back d'une application Spring Boot
Code Source sur repo Github : https://github.com/ImFlog/microservices-security

Les développeurs Keycloak fournissent une librairie Java compatible avec Spring Security : classe *KeycloakPreAuthActionsFilter*, *KeycloakAuthentificationProcessingFilter* …

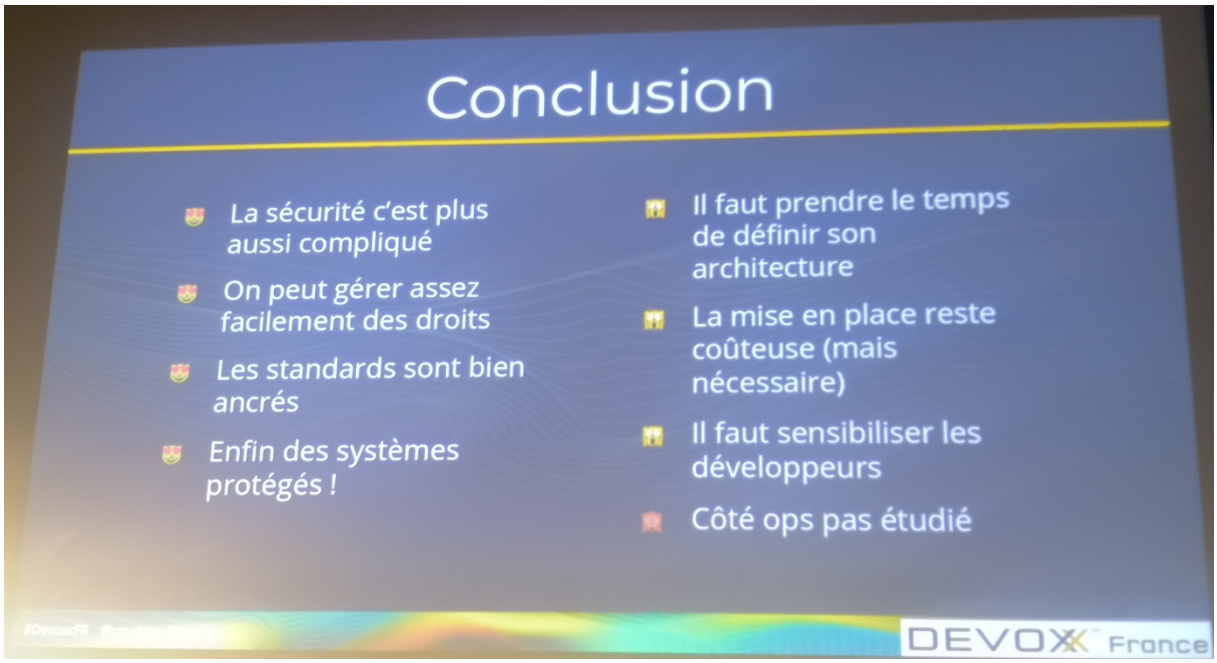
L'API REST Spring MVC est sécurisée avec l'annotation
@PreAuthorize("hasRole('ROLE_yes_we_can'")

Pour propager le token JWT d'une application Spring Boot à une application .NET, il est nécessaire d'extraire manuellement le token JWT de l'entête HTTP pour le passer au RestTemplate.

Lors du 1er appel, il y'a un temps de Warmup car le serveur doit aller chercher la clé publique sur Keycloack afin de valider le jeton JWT. Cette clé est mis en cache.

Conclusion

En conclusion, sécuriser une architecture n'est pas si compliquée : on s'appuie sur des briques existantes. Cela demande néanmoins une réflexion en amont pour sa mise en place. Il faut également former les développeurs pour propager le header HTTP d'autorisation.


Conclusion
La sécurité c'est plus aussi compliqué
On peut gérer assez facilement des droits
Les standards sont bien ancrés
Enfin des systèmes protégés !
Il faut prendre le temps de définir son architecture
La mise en place reste coûteuse (mais nécessaire)
Il faut sensibiliser les développeurs
Côté ops pas étudié
DEVOXX France