Authentification

Plan et objectifs

- Identification, authentification et autorisation
- Authentification HTTP de base
- Gestion des mots de passe dans une application
- Hachage et salage
- Gestion de sessions
  - Identificateur de session
  - JWT - JSON Web Token
- OAuth 2 (3-étapes et 2-étapes)

Identification

Chaque utilisateur reconnu d'un système (humain ou autre système) possède un identifiant.

Cet identifiant permet de déterminer quels sont les permissions et accès que cet utilisateur possède.

Typiquement, le nom d'utilisateur (ou courriel) est l'identifiant d'un utilisateur dans une application web.

Authentification

Il ne suffit pas de seulement s'identifier pour pouvoir avoir accès à l'application web, il faut aussi **prouver que nous sommes bien la personne qui possède l'identifiant**.

Ainsi, pour accéder à une application web, on doit fournir l'identifiant (nom d'utilisateur) et une information secrète (e.g. mot de passe) connue seulement de l'individu qui possède l'identifiant.

Pour une sécurité accrue, il est possible d'utiliser une **authentification à multiples facteurs**. Dans ce cas, l'utilisateur doit fournir plus d'une preuve pour obtenir l'accès.

Idéalement, les multiples **preuves** demandées sont **de natures différentes**:

- ce qu'il sait: mot de passe
- ce qu'il possède: code sur téléphone
- ce qu'il est: biométrie

Autorisation

Une fois l’étape d’authentification complétée, le système applique les règles d’accès et les permissions assignées à l’utilisateur en question.

Ce mécanisme se nomme l’autorisation.

L’autorisation détermine les ressources que l’utilisateur peut consulter et les opérations qu’il peut effectuer dans le système.

Une question de risque

L’ajout d'un processus d’authentification amène de la friction pour les utilisateurs. Plus le processus est sécuritaire (e.g. multi-facteurs) plus la friction est élevée.

Il faut donc ajuster le processus d’authentification en fonction des ressources qui sont protégées.

Plus la valeur des ressources protégées est élevée:

- plus les méthodes d’authentification doivent être fortes
  - double facteur
  - politique des mots de passe
- plus les autorisations doivent être restrictives
  - principe de moindre privilège

Encodage Base64 et Base64Url

Méthode pour encoder n'importe quelle suite d'octets en une séquence de caractères selon la table de codage du Base64.

Pour ce faire, à partir de la suite d'octets initiale, pour chaque 3 octets, on les divise en 4 nombres de 6 bits chacun. Chaque nombre est alors remplacé par un caractère défini par la table de codage.

S'il manque des octets lors du groupement, on ajoute des paires de 0s pour compléter le dernier groupe et on ajoute un caractère '=' par paires de 0s ajoutées dans la chaîne encodée pour savoir combien de 0 enlever lors du décodage.

a*b***c**d -> 0110 0001 *0110 0010* **0110 0011** 0110 0100 -> 011000 01*0110 0010***01 100011** 011001 00*0000* -> 24 22 9 35 25 0 -> **YWJjZA==**

Notez que la taille du message encodée est supérieure au message initial.

Base64Url remplace les caractères + et / par - et _ pour permettre d'utiliser le résultat dans un URL ou comme nom de fichier.

https://fr.wikipedia.org/wiki/Base64

Authentification HTTP de base (*HTTP basic auth*)

Dans ce mécanisme d'authentification, le client envoie le nom d'usage et le mot de passe séparé par un ":" puis encodé en base64.

Le résultat est mis dans l'en-tête `Authorization`.

`alice:motdepasse` devient `YWxpY2U6bW90ZGVwYXNzZQ==`

Puis, on ajoute l'en-tête ci-dessous à la requête HTTP **pour chaque requête**.

```
Authorization: Basic YWxpY2U6bW90ZGVwYXNzZQ==
```

Notez qu'un message encodé n'est pas crypté! Il faut donc utiliser HTTPS.

Transmission du mot de passe

Lorsque l'utilisateur entre son nom d'utilisateur et son mot de passe dans un formulaire web, l'application doit pouvoir transmettre ces informations sur une connexion vers le backend.

La façon la plus simple est d'utiliser une connexion sécurisée (HTTPS).

HTTPS garantit qu'on envoie les informations au bon serveur (certificat) et que les données sont cryptées lors de l'échange.

Gestion des mots de passe

Le mot de passe est une information que l'utilisateur connaît et que le système doit pouvoir valider.

Naïvement, on pourrait imaginer stocker le mot de passe dans une base de données, mais si un attaquant réussit à accéder à la base de données, les mots de passe seront accessibles.

Sachant que les mots de passe sont souvent réutilisés (malheureusement), cette fuite de données peut avoir des répercussions au-delà du système en question.

Il faut donc trouver une façon de stocker une information qui nous permettra de valider le mot de passe sans jamais stocker le mot de passe en clair.

Interlude Crypto 1 - Fonction de hachage

*On nomme **fonction de hachage**, une fonction particulière qui, à **partir d'une donnée fournie en entrée, calcule une empreinte numérique servant à identifier rapidement la donnée initiale, au même titre qu'une signature** pour identifier une personne. Les fonctions de hachage sont utilisées en informatique et en cryptographie notamment pour reconnaître rapidement des fichiers ou des mots de passe.*

Une fonction de hachage nous permet donc, à partir du mot de passe, de calculer simplement un *hash* (chaîne de caractères) *unique.*

Une fonction de hachage est à sens unique, ce qui signifie que le calcul [mot de passe -> *hash*] est rapide, mais que le calcul inverse [hash -> mot de passe] est impossible.

Encodage, Hachage et Chiffrement

Encodage

- Conversion d'un message selon un algorithme précis
- Opération bidirectionnelle sans utilisation de secret
- Aucune sécurité

Hachage

- Génération d'un *hash* qui varie selon l'entrée de l'algorithme de hachage
- Opération unidirectionnelle (à partir du *hash*, on ne peut pas trouver l'entrée)
- Utilisé pour des signatures (e.g. vérifier que le contenu n'a pas été modifié)

Chiffrement

- Conversion d'un message selon un algorithme précis
- Opération bidirectionnelle, mais nécessite un *secret* pour décrypter
- Sécuritaire

Hachage et mots de passe

Comme l'application de la fonction de hachage est rapide et unidirectionnel, on applique la fonction de hachage au mot de passe que l'on peut alors stocker.

À partir de ce moment, il suffit d'appliquer à nouveau la fonction de hachage sur le mot de passe reçu lors de la connexion et de comparer avec la valeur stockée dans la base de données pour le valider.

Si jamais la base de données est compromise, seuls les *hash* seront visibles et comme il est *impossible* de retrouver les mots de passe à partir du *hash*, l'attaquant n'a pas accès aux mots de passe de vos utilisateurs (mais il a quand même eu accès à votre DB 😬).


IMPOSSIBLE TU DIS?
HACKERMAN
imgflip.com

Attaque avec une *table arc-en-ciel* 🌈

Si on a accès aux *hashs* et que l'on voit deux *hashs* identiques qu'est-ce qu'on peut déduire?

Et si l'attaquant a précalculé une table (ordonnée par hash) des mots de passe les plus courants?

nom d'utilisateur	hash
Alice	d41d8cd98f00b204e9800998ecf8427e
Rob	964d72e72d053d501f2949969849b96c
Bob	d41d8cd98f00b204e9800998ecf8427e

mot de passe	hash
123456	d41d8cd98f00b204e9800998ecf8427e
motdepasse	e72d053964d72d501f2949969849b96c
licorne	964d72e72d053d501f2949969849b96c

Le salage

Pour ralentir et dissuader les attaquants, on ajoute un sel (salt) lorsque que l'on applique la fonction de hachage sur le mot de passe.

Le sel est une chaîne de caractères générées aléatoirement pour chaque utilisateur et qui est stockée dans la base de données.

Maintenant, la formule devient: *hash(mot de passe + sel)*

Il reste possible que l'attaquant trouve un mot de passe avec une stratégie force brute, mais il devra recalculer individuellement pour chaque utilisateur.

Fonction de hachage pour les mots de passe

Pour les mots de passe, on désire une fonction de hachage *lente* et *coûteuse* pour dissuader l'attaquant.

Quelques algorithmes de hachage recommandés pour les mot de passe: Argon2, Bcrypt, PBKDF2.

https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

Mécanisme d'authentification

Pour transmettre les informations d’authentification au serveur, il existe différentes options:

- l’en-tête HTTP Authorization
  - Basic: le nom d’utilisateur et le mot de pass est retransmis à chaque requête
  - Bearer: le jeton (token) est retransmis à chaque requête
- les cookies HTTP
  - les cookies sont initialement créés par le serveur
  - les cookies sont retransmis à chaque requête pour la même origine

Cookie

Il est possible de modifier le comportement de base pour l'utilisation des cookies côté client.

Quelques attributs optionnels des cookies:

- **`Path`** - Détermine un fragment d'URL qui doit être présent dans l'URL de la requête pour que le cookie sera retransmis.

  ```
  Set-Cookie: mykey=myvalue; Path=/
  ```

- **`Domain`** - Pour quels domaines (sous-domaines) le cookie doit être transmis. Par défaut, seulement pour le même *host*.

  ```
  Set-Cookie: mykey=myvalue; Domain=example.org
  ```

  (inclus domaine et sous-domaine comme app.example.org)

Cookie (2)

- **`Expires`** - Détermine le moment où le cookie expire.

  ```
  Set-Cookie: mykey=myvalue; Expires=Thu, 21 Oct 2021 07:28:00 GMT
  ```

- **`Max-Age`** - Nombre de secondes avant l'expiration.

  ```
  Set-Cookie: mykey=myvalue; Max-Age=600
  ```

- **`Secure`** - Indique que le cookie n'est transmis que si la connexion est sécurisée (HTTPS).

  ```
  Set-Cookie: mykey=myvalue; Secure
  ```

- **`HttpOnly`** - Le cookie n'est pas accessible à partir du code javascript.

  ```
  Set-Cookie: mykey=myvalue; HttpOnly
  ```

- **`SameSite`** - Détermine si le cookie peut être envoyé vers une autre origine. Il existe 3 valeurs possibles: Strict (seulement vers l'origine), Lax (comme Strict mais aussi sur la navigation vers l'origine), None (envoyé sur différentes origines, mais requiert `Secure`).

  ```
  Set-Cookie: mykey=myvalue; SameSite=Strict
  ```

Gestion de sessions avec les cookies

Une fois que l'utilisateur s'est connecté (login), on veut pouvoir valider que les requêtes subséquentes seront autorisées par le serveur sans avoir à retransmettre les informations de connexions (nom d'utilisateur et mot de passe).

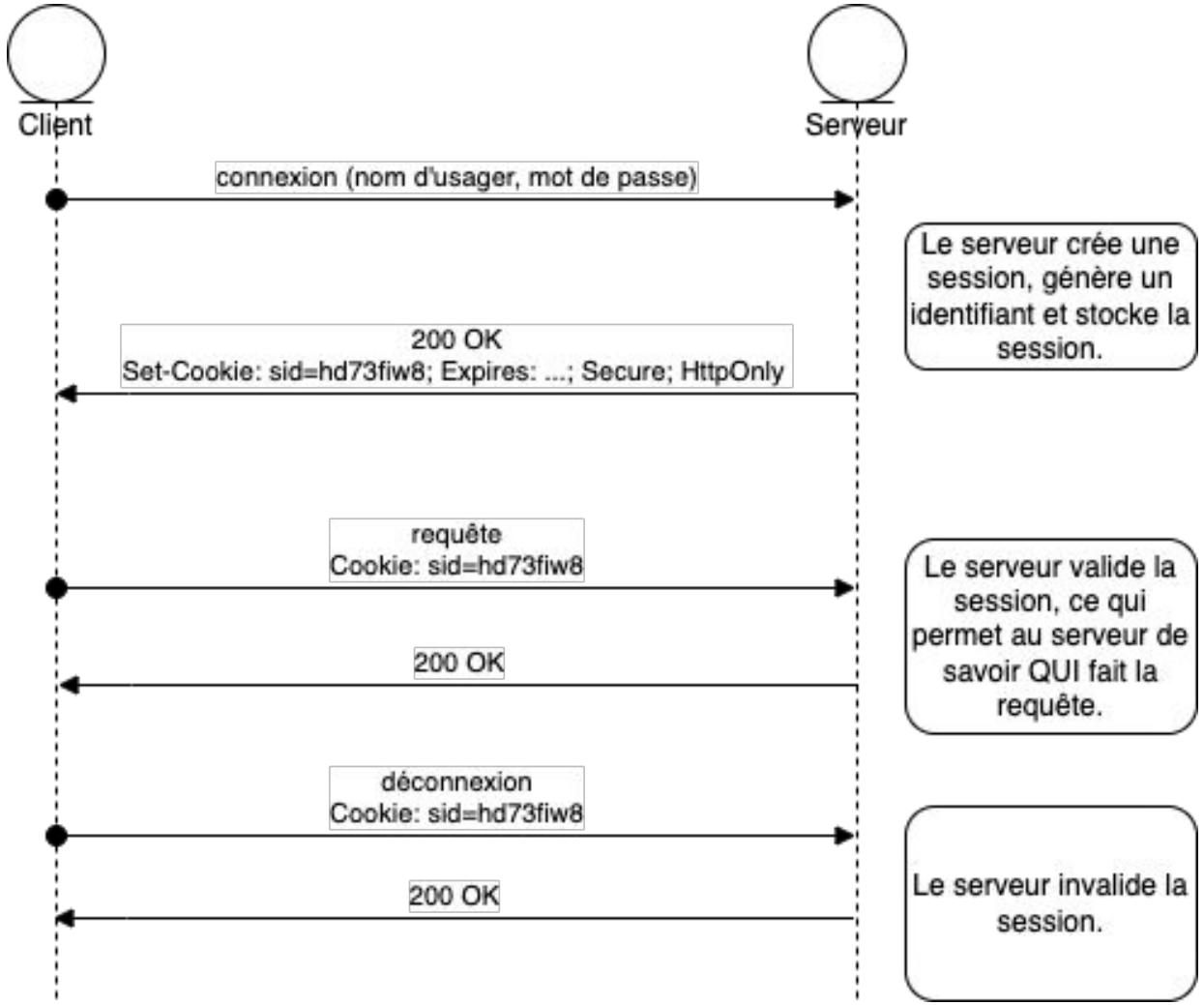
À la base le protocole HTTP est sans état, mais l'utilisation des cookies permet au serveur de donner une information qui lui sera rendue à chaque requête.

Une option consiste à créer une session du côté serveur suivant une connexion réussie. Cette session représente le fait que l'utilisateur s'est bien authentifié.

En réponse à la connexion réussie, le serveur ajoute un cookie qui contient un identifiant ou un jeton pour la session active (token).

Ce jeton permettra au serveur de vérifier si la session existe lors des requêtes subséquentes.

À la déconnexion, le serveur pourra supprimer cette session pour indiquer que l'utilisateur n'a plus de session valide et qu'il doit se reconnecter.


Client
Serveur
connexion (nom d'usager, mot de passe)
Le serveur crée une session, génère un identifiant et stocke la session.
200 OK
Set-Cookie: sid=hd73fiw8; Expires: ...; Secure; HttpOnly
requête
Cookie: sid=hd73fiw8
Le serveur valide la session, ce qui permet au serveur de savoir QUI fait la requête.
200 OK
déconnexion
Cookie: sid=hd73fiw8
Le serveur invalide la session.
200 OK

Bonnes pratiques

- Ne pas utiliser un nom de cookie qui pourrait indiquer la technologie utilisée (e.g. springbootsid=yyyzzz)
- Pour limiter les attaques par force brute, choisir un identifiant **aléatoire** d'au moins 128 bits (16 caractères).
  - Force brute = L'attaquant génère une multitude d'identificateurs en espérant tomber sur un qui fonctionne.
- L'identifiant ne doit pas contenir d'information sensible visible.
- Toujours mettre une expiration avec `Expires` ou `Max-Age`.
- Utiliser `Secure` et `HttpOnly`

https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

Interlude Crypto 2 - Cryptographie asymétrique

Lorsqu'on utilise une méthode de cryptographie asymétrique, on doit posséder 3 éléments:

- une fonction
- une clé publique
- une clé privée

La clé publique est publique (!).

On utilise cette clé publique et la fonction pour crypter le message.

Seule l'utilisation de la clé privée permet de décrypter le message.

HMAC : *hash-based message authentication code*

Mécanisme qui consiste à créer un hash en choisissant une fonction de hachage et une clé secrète connue seulement de l'émetteur.

Cette méthode est utilisée pour signer des messages.

On applique la fonction de hachage en utilisant le message et la clé secrète comme entrée et la fonction nous donne un code (HMAC) en sortie (signature).

On ajoute ensuite la signature au message original.

Quiconque possède la clé secrète peut valider l'intégrité du message, car si le message a été modifié, la signature ne correspondra plus et le message sera rejeté.

Cette méthode ne crypte pas le message!

JSON Web Token - JWT (jot?!)

Un autre mécanisme populaire pour gérer l'authentification est le JSON Web Token (JWT). Contrairement au mécanisme précédent utilisant les cookies de session, le serveur n'a pas à créer ni conserver de session.

Le JWT contient l'ensemble des informations nécessaires, incluant l'identifiant de l'utilisateur.

Le jeton est signé par le serveur.

Le client renvoie le jeton dans ses requêtes (en-tête `Authorisation`, cookie).

Le serveur peut alors valider la signature pour s'assurer que le jeton n'a pas été modifié.

Structure du JWT

Le JWT est constitué de 3 parties:

- en-tête (header): Fragment JSON qui détermine l'algorithme et le type du jeton (encodé en Base64Url)
- charge utile (payload): Fragment JSON contenant les données (encodé en Base64Url)
- signature: une chaîne de caractères qui prouve que le contenu du message n'a pas été modifié

Le jeton est construit en concaténant les informations: xxx.yyy.zzz.

Exemple de jeton:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

JWT: en-tête

L’en-tête ne contient généralement que deux champs:

- **`alg`**: algorithme utilisé pour la signature
- **`typ`**: le type de jeton, typiquement *jwt*

JWT: Charge Utile

La charge utile contient plusieurs attributs appelés *claims*.

Il existe trois types de claims:

- Enregistrés: Attributs faisant partie du standard.
  - **`iss`** (issuer: le générateur du jeton),
  - **`sub`** (subject: l'utilisateur),
  - **`aud`** (audience: le serveur de ressources),
  - **`exp`** (expiration: date d'expiration),
  - **`iat`** (issue at: date de création).
- Publiques: Attributs ne faisant pas partie du standard, mais communs.
  - voir https://www.iana.org/assignments/jwt/jwt.xhtml
- Privés
  - Champs spécifiques à l'application (e.g. role, admin, etc).

Les claims sont généralement utilisés pour stocker les informations qu'une session pourrait contenir.

Mécanisme de signature

Pour permettre au serveur de vérifier que le jeton reçu du client n'a pas été altéré, la troisième partie du jeton est une signature cryptographique.

```
Algo(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

Différents algorithmes peuvent être utilisés. Soit des fonctions de hachage pour générer un HMAC, soit un algorithme de cryptage asymétrique.

https://jwt.io/

JWT bonnes pratiques

Attention! Le contenu du JWT n'est qu'encodé donc n'importe qui peut lire l'information qu'il contient, sauf s'il est crypté avant d'être encodé en Base64Url.

Il est donc important de ne pas mettre des informations sensibles dans le contenu du jeton.

Pour transmettre un JWT on peut utiliser soit l'en-tête `Authorization` ou les cookies.

Si on utilise les cookies, on veut utiliser les attributs `Secure` et `HttpOnly` pour éviter qu'un scripte malhonnête n'accède au jeton et l'utilise pour effectuer des requêtes.

Si on utilise l'en-tête `Authorization`, il est préférable de garder le JWT en mémoire.

Il est possible de le garder dans le localStorage mais cela le rend vulnérable à certains types d'attaques.

Et à la déconnexion?

Un des avantages des JWTs est que le serveur n'a pas l'obligation de conserver les données de session car ces données sont soit mises dans le jeton, soit le jeton contient les données nécessaires pour les récupérer (JWT = serveur potentiellement sans état).

Mais au moment de la déconnexion, comment peut-on invalider les jetons?

Options pour la déconnexion

Il est impossible de révoquer un jeton globalement, plutôt:

- nettoyer le jeton côté client en réponse au logout
- utiliser un délai d'expiration *court*

Et si l'utilisateur change son mot de passe et qu'on veut invalider tous ses autres jetons?

- Utiliser une liste noire qui contient la liste des jetons qui ont été annulés dans une base de données
- Nettoyer cette liste lors de l'expiration normal du jeton
- Vérifier à chaque requête si le jeton fait partie de la liste noire et si oui, la requête est rejetée.

OAuth 2

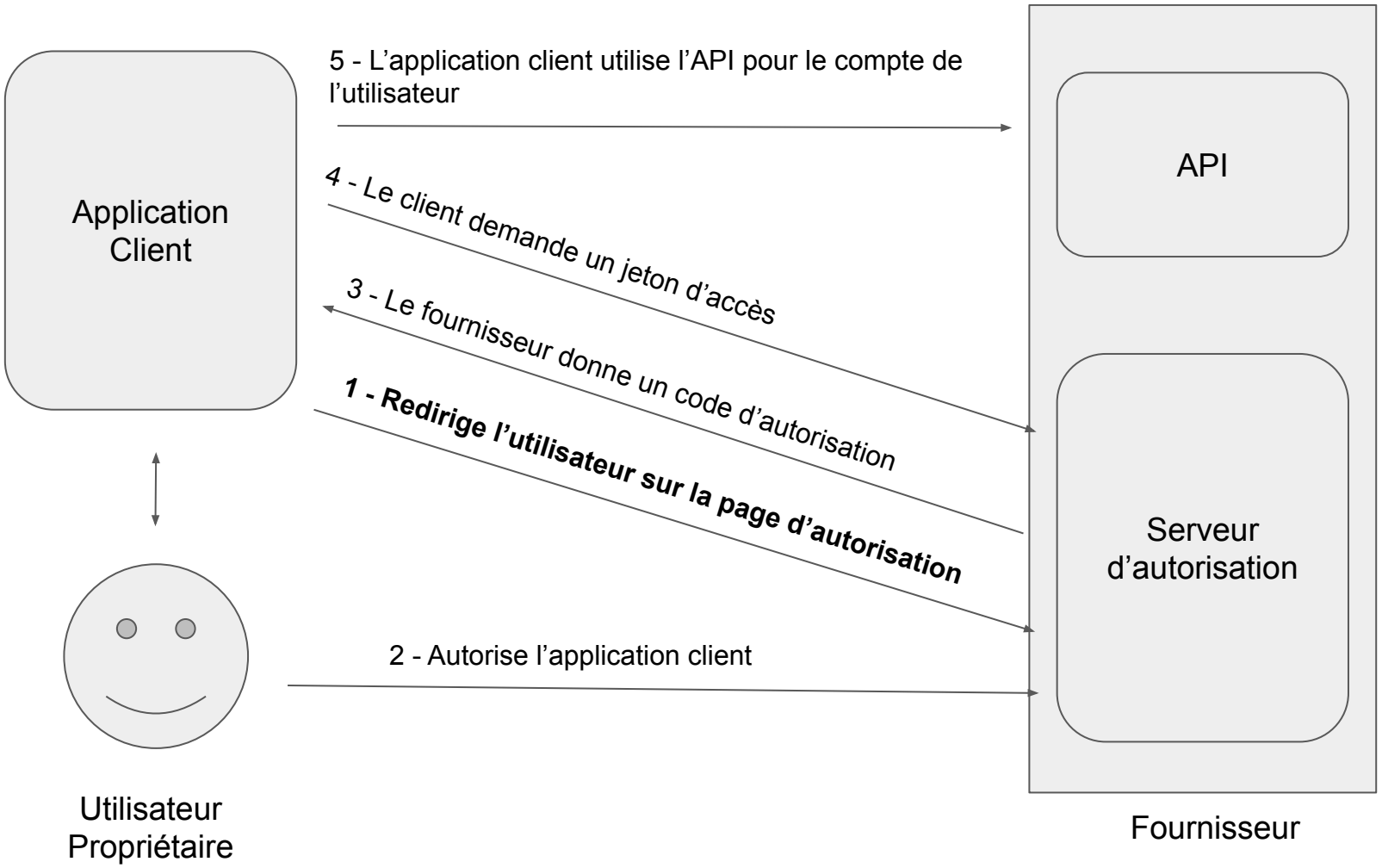
OAuth 2 est à la base un **protocole de délégation d'autorisation**.

Il permet à une application d'accéder aux ressources protégées d'un autre fournisseur pour le compte d'un utilisateur.

Par exemple, si vous voulez créer une application qui se connecte aux APIs d'autre fournisseur (e.g. LinkedIn, Github, etc) pour accéder à des données de vos utilisateurs, vous allez devoir utiliser OAuth.

L'avantage est que l'utilisateur n'aura pas à vous donner ses informations de connexion. Cette étape sera déléguée au fournisseur, qui vous donnera ensuite un jeton à utiliser pour que votre application se connecte à ses APIs.

OAuth 2 - Les participants



OAuth 2 - Séquence

Au départ, l'application client doit s'enregistrer chez le fournisseur pour obtenir un identifiant et un secret (*client id et client secret*).

Ensuite, pour accéder aux ressources du fournisseur:

1. L'application client redirige l'utilisateur vers la page d'autorisation du fournisseur et ajoute son *client id* et un URL de retour aux paramètres de la requête.
2. L'utilisateur autorise l'application client.
3. Le fournisseur redirige l'utilisateur vers le URL de retour qui contient un code d'autorisation temporaire dans les paramètres de requête.
4. L'application doit alors échanger ce code d'autorisation contre un jeton d'accès en fournissant aussi son *client id* et son *client secret*.
5. L'application peut alors utiliser le jeton d'accès (en-tête `Authorization Bearer`) pour utiliser l'API pour le compte de l'utilisateur.

Exemple LinkedIn - https://learn.microsoft.com/en-us/linkedin/shared/authentication/authorization-code-flow

OAuth 2 - 2 étapes

Dans les cas où l'application client veut accéder à des ressources d'un fournisseur, mais que ces ressources ne sont pas spécifiques à un utilisateur en particulier, le processus peut être simplifié.

Dans ce cas, l'application client:

1. Demande un jeton d'accès directement en fournissant son *client id* et son *client secret*.
2. Le fournisseur donne un jeton d'accès.
3. Le client utilise le jeton d'accès pour utiliser l'API du fournisseur.

Exemple LinkedIn - https://learn.microsoft.com/en-us/linkedin/shared/authentication/client-credentials-flow