BestCoder

Comprendre OAuth 2.0 par l'exemple

12 août 2019

Table des matières

OAuth 2.0, successeur du protocole OAuth 1.0a, est **un framework d'autorisation** permettant à une application tierce d'accéder à un service web.

Largement utilisé dans le domaine du web avec notamment Facebook ou encore Google, OAuth est devenu incontournable.

En tant que développeur, nous pouvons être amenés à utiliser un serveur fournissant un accès via OAuth 2.0 ou à implémenter un serveur d'autorisation pour sécuriser une API en utilisant ce framework.

Nous allons donc voir les concepts nécessaires pour comprendre et utiliser sciemment le framework OAuth 2.0. Nous n'allons cependant pas explorer les détails techniques liés à OAuth 2.0. L'objectif est de comprendre comment fonctionne OAuth 2.0 et pas comment l'implémenter.

Voici les prérequis pour suivre cet article :

— Connaissances en développement web et sur le protocole HTTP ;
— Connaissances sur les API.

1. La naissance de OAuth

Formalisé dans la RFC 6749 en octobre 2012, OAuth 2.0 est né d'un besoin simple.

? Comment une application peut-elle accéder à une ressource protégée au nom de son propriétaire sans connaitre ses identifiants (login et mot de passe) ?

Avec la démocratisation des applications web et la prolifération des applications mobiles, différentes applications sont amenées à interagir entre elles. Ainsi, un site web A pourrait utiliser les données d'un réseau social connu afin d'inscrire un utilisateur.

Cette démarche permet à l'utilisateur de donner accès à ses informations personnelles déjà disponibles sur le réseau social au site web A.

Avant l'arrivée de OAuth, l'utilisateur devait faire confiance au site web A et lui donner ses identifiants.

Cependant, pour éviter de redemander les identifiants à l'utilisateur, les applications tierces avaient tendance à **conserver le mot de passe de celui-ci en clair**.

En plus des limites de sécurité liées à cette conservation du mot de passe en clair, l'application tierce était en mesure **d'accéder à toutes les informations protégées sans distinction**.

Et hormis ces problèmes de sécurité, une question d'ordre pratique se pose.

En effet, si l'utilisateur a fourni ses identifiants à plusieurs applications tierces et décide de les changer, il doit le **modifier aussi pour toutes ces applications**.

Pour répondre à ces problématiques, OAuth 2.0 a été créé afin de permettre des interactions entre applications dans un contexte sécuritaire optimal. OAuth 2.0 a été conçu pour être utilisé avec le protocole HTTPS.

Pour ceux connaissant déjà OAuth 1.0a (standardisé en 2010), OAuth 2.0 se différencie en partie de son prédécesseur par le fait que l'aspect sécuritaire lié à l'intégrité et la confidentialité des données transmises pendant la demande d'autorisation est délégué au protocole TLS .

> **!** OAuth 2.0 doit donc toujours s'utiliser avec une connexion sécurisée (HTTPS) pour bien remplir son objectif.

2. Les roles

Afin de bien comprendre les mécanismes en jeu lors d'une demande d'autorisation avec OAuth 2.0, nous devons d'abord définir quelques notions utilisées dans les spécifications de ce framework.

Le schéma *classique* d'un processus d'authentification et d'autorisation fait intervenir deux parties :

— **un serveur** en mesure d'authentifier et d'autoriser l'accès à une ressource ;
— **un utilisateur** qui fournit ses identifiants pour accéder à la ressource.

> **i** La ressource désigne ici toute information appartenant à l'utilisateur et qui est exposée par le serveur (son identité, ses photos, ses messages, etc.).

Pour OAuth, le processus d'autorisation est un peu plus complexe et peut faire intervenir jusqu'à quatre acteurs.

2.0.1. Propriétaire de la ressource : *Resource owner*

Le propriétaire de la ressource est une entité (par exemple un utilisateur) en mesure de donner l'accès à une ressource protégée.

2.0.2. Client

Le client désigne l'application tierce qui demande l'accès à la ressource au nom de son propriétaire. Le terme client peut parfois induire en erreur car il peut désigner, ici, aussi bien une application mobile qu'un serveur web. Il permet juste d'identifier l'entité qui souhaite accéder à une ressource.

2.0.3. Serveur de ressource : *Resource server*

Le serveur de ressource désigne le serveur qui héberge les ressources protégées.

2.0.4. Serveur d'autorisation : *Authorization server*

Le serveur qui délivre le droit d'accès à la ressource protégée au client après avoir authentifier le propriétaire de la ressource.

Dans les faits, le serveur d'autorisation et le serveur de ressource sont souvent confondus. Ainsi, un même serveur pourra jouer le rôle de serveur d'autorisation et héberger les ressources protégées.

A titre d'exemple, si un site web nous demande notre autorisation pour publier sur notre mur Facebook, nous jouons le rôle de propriétaire de la ressource, le site web est le client et le serveur de Facebook joue le double rôle de serveur d'autorisation et de serveur de ressource.

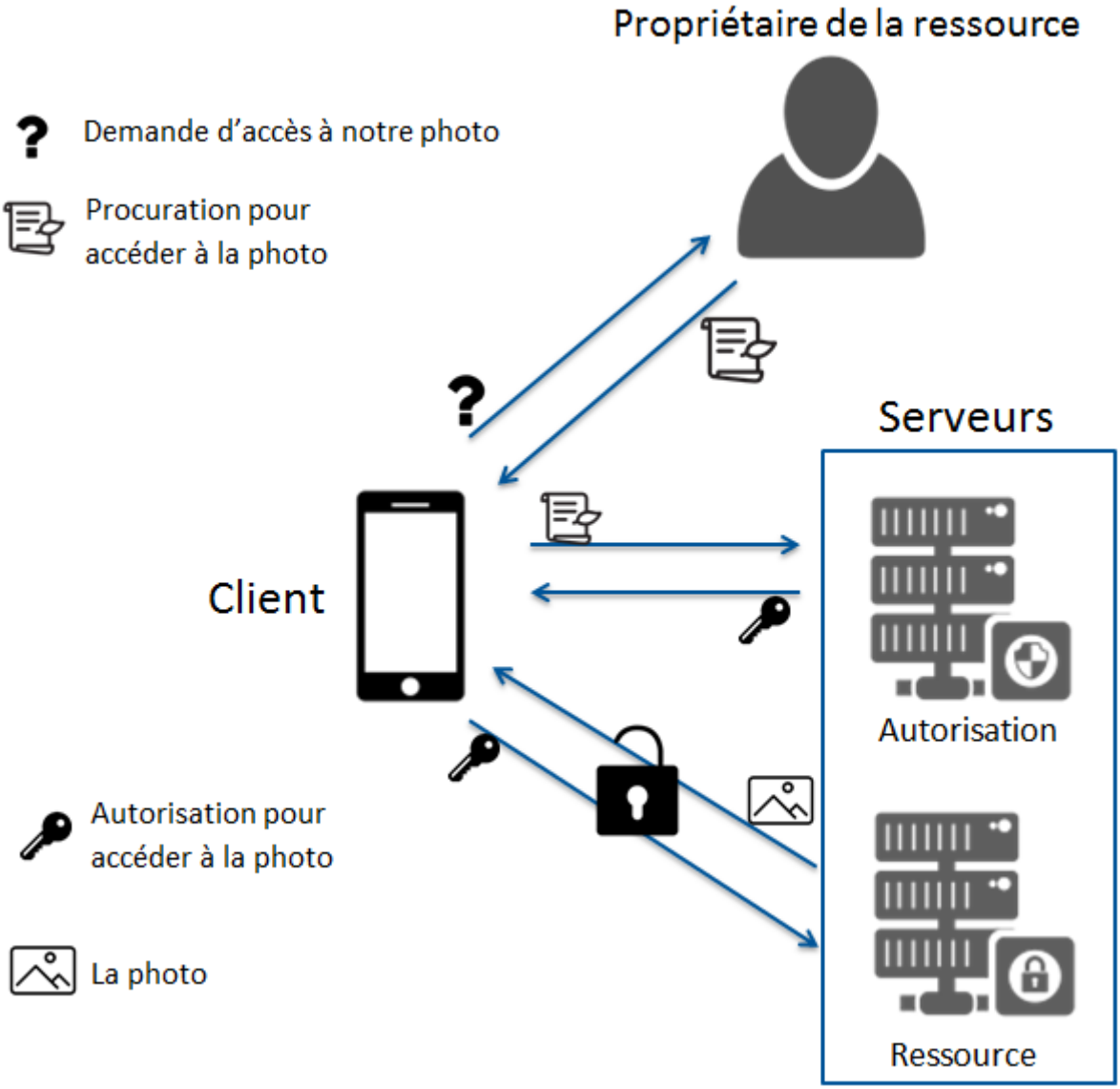

FIGURE 2. – Les rôles dans OAuth 2.0

3. La gestion des clients

Une demande d'autorisation avec OAuth est toujours initiée par un client. Pour tous les clients, il faudra donc les enregistrer auprès du serveur d'autorisation.

L'enregistrement nécessite au moins trois informations :

— l'identifiant du client ;
— le mot de passe ou la paire de clés (publique/privée) pour les clients confidentiels ;
— et une ou plusieurs URL de redirection.

Bien que les spécifications de OAuth 2.0 n'excluent pas l'utilisation de clients non-enregistrés, leur utilisation est au-delà des spécifications du protocole. Nous n'aborderons donc pas ce cas d'usage.

4. Les tokens

La demande d'accès à une ressource protégée via OAuth se traduit par la délivrance d'un *token* au client. Le *token* représente juste une chaîne de caractère unique permettant d'identifier le client et les différentes informations utiles durant le processus d'autorisation.

Le serveur d'autorisation est en mesure d'en fournir deux types.

4.0.1. Token d'accès : *Access token*

Le *token* d'accès permet au client d'accéder à la ressource protégée. Ce *token* a une durée de validité limitée et peut avoir une portée limitée.

Cette notion de portée permet d'accorder un accès limité au client. Ainsi, un utilisateur peut autoriser un client à accéder à ses ressources qu'en lecture seule.

4.0.2. Token de rafraîchissement : *Refresh token*

Le *token* de rafraîchissement permet au client d'obtenir un nouveau *token* d'accès une fois que celui-ci a expiré. Sa durée de validité est aussi limitée mais est beaucoup plus élevée que celle du *token* d'accès.

Son utilisation permet au client d'obtenir un nouveau *token* d'accès sans l'intervention du propriétaire de la ressource protégée.

En résumé, OAuth 2.0 formalise un ensemble de mécanismes permettant à une application tierce (client) d'accéder à une ressource protégée au nom de son propriétaire (*resource owner*) ou en son propre nom. Cette autorisation se traduit par la délivrance d'un *token* d'accès (et éventuellement d'un *token* de rafraîchissement) qui permet au client de dialoguer avec le serveur hébergeant les ressources protégées (serveur de ressource).

5. Les différents scénarios d'autorisation

Contrairement à son prédécesseur qui ne proposait qu'un seul scénario d'autorisation, le framework OAuth 2.0 a été spécifié avec quatre méthodes différentes pour obtenir une autorisation. En plus, le framework est extensible et permet donc de rajouter autant de méthodes que nous le souhaitons.

Selon la nature du client et du niveau d'accès souhaité, le scénario à utiliser n'est pas forcément le même. Le serveur d'autorisation peut ainsi autoriser l'utilisation d'un ou de plusieurs scénarios selon les besoins.

5.1. Autorisation avec un code : *Authorization Code Grant*

Ce processus d'autorisation principalement utilisé par les clients confidentiels permet d'obtenir un *token* d'accès (*Access token*) et un *token* de rafraîchissement (*Refresh token*). Il nécessite l'intervention du client, du propriétaire de la ressource protégée et du serveur d'autorisation. C'est d'ailleurs le mode d'utilisation à l'origine du protocole OAuth 1.0.

Les clients dits **confidentiels** sont les clients en mesure de conserver en toute confidentialité les identifiants qui leurs sont affectés.

5.1.1. Cas d'usage

Le cas d'usage typique pour ce type d'autorisation est lorsque le client est un code coté serveur.

Si par exemple nous développons une application web en PHP et que **depuis le serveur** nous voulions accéder à certains services Google (profil Google Plus, mails, etc.) d'un utilisateur, le procédé d'autorisation avec un code serait le plus approprié. L'accès au code serveur étant sécurisé, le client peut être considéré comme étant confidentiel.

5.1.2. Description

Le processus d'autorisation se déroule comme suit :

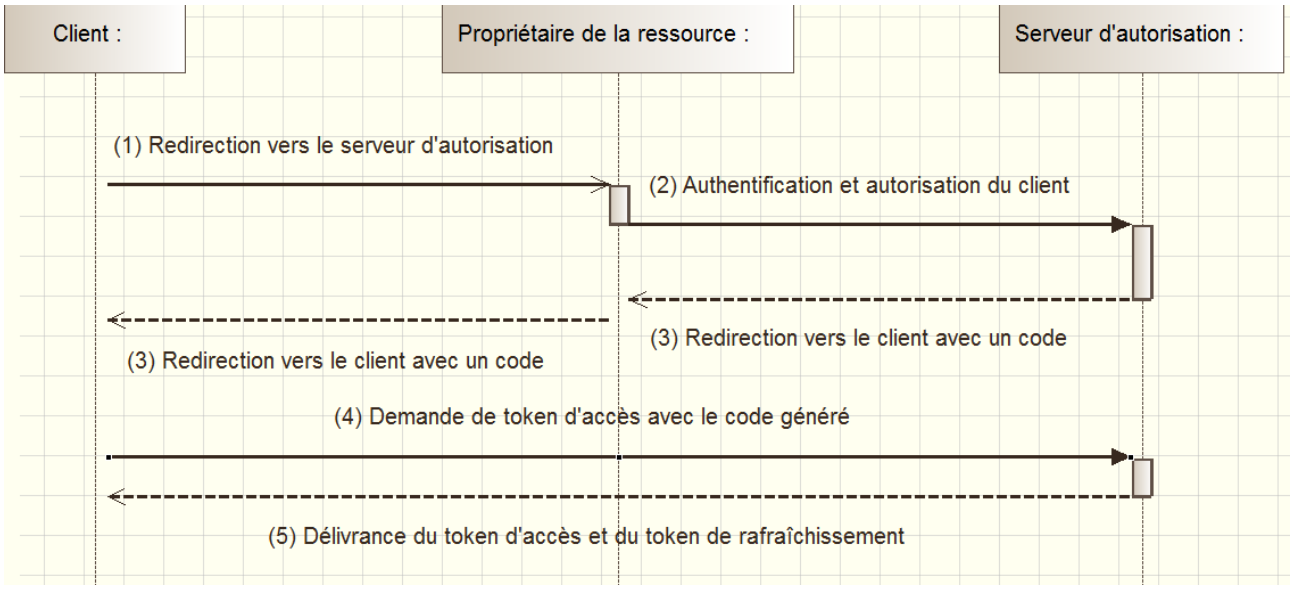

FIGURE 5. – Authorization Code Grant

1. Le client redirige le propriétaire de la ressource vers le serveur d'autorisation. Le client doit inclure son identifiant dans la requête de redirection et le niveau d'accès qu'il souhaite obtenir.
2. Le propriétaire de la ressource s'authentifie auprès du serveur d'autorisation et approuve ou non la requête du client.

3. Si la requête est autorisée, le serveur d'autorisation redirige à nouveau le propriétaire de la ressource en utilisant l'URL de redirection défini par le client. Cette URL est renseignée à l'enregistrement du client et peut aussi être choisie dans la requête à l'étape 1. La requête de redirection contient un code d'autorisation dans l'URL.
4. Avec le code d'autorisation ainsi obtenu, le client demande un *token* d'accès en prenant le soin de s'authentifier à son tour auprès du serveur d'autorisation.
5. Une fois le client authentifié, le serveur d'autorisation valide le code d'autorisation et s'assure que l'URL de redirection est identique à celle utilisée dans la troisième étape. Si toutes ces contraintes sont respectées, le serveur d'autorisation renvoie au client un *token* d'accès et éventuellement un *token* de rafraîchissement.

Pour assurer ces différentes interactions, le serveur d'autorisation doit mettre à disposition des clients deux URL. Une URL d'autorisation (Authorization endpoint) qui sera utilisée dans l'étape 1 et permettra d'obtenir un code d'autorisation et une URL de génération de *tokens* (Tokens endpoint) permettant d'obtenir les différents *tokens*.

5.1.3. Exemple d'utilisation

Pour mieux illustrer ces propos, nous allons prendre un exemple simple. Nous développons un site web appelé **example.com** permettant aux utilisateurs de sauvegarder leurs emails importants sur Gmail.

Pour ce faire, notre site doit pouvoir accéder à ses informations au nom de l'utilisateur.

Nous devons donc commencer par enregistrer notre application auprès des services de Google avec un identifiant (client_id), un mot de passe (client_secret) et une URL de redirection (redirect_uri).

Pour les besoins de cet exemple, nous prendrons comme identifiant `id_1`, comme mot de passe `secret_1` et comme URL de redirection `example.com/oauth2-redirect`.

Google expose deux *endpoints* pour gérer ce type d'autorisation :

— https ://accounts.google.com/o/oauth2/v2/auth pour initialiser le processus d'autorisation (par exemple, la génération de code d'autorisation) ;
— et https ://www.googleapis.com/oauth2/v4/token pour la génération des *tokens* d'accès.

Le processus commence par une redirection de l'utilisateur (propriétaire de la ressource) vers l'URL https ://accounts.google.com/o/oauth2/v2/auth ?scope=email&state=utile_pour_eviter_les_failles_csrf&redirect_uri=https%3A%2F%2Fexample.com%2Foauth2-redirect&response_type=co **(Étape 1)**.

Si l'utilisateur s'authentifie (ou est déjà authentifié) et autorise notre application **(Étape 2)**, Google le redirige vers notre site avec le code d'autorisation.

L'URL de redirection est utilisée ainsi pour transmettre les informations à notre site https ://example.com/au redirect ?code=code_autorisation&state=utile_pour_eviter_les_failles_csrf **(Étape 3)**.

Notre site peut maintenant demander un *token* d'accès en utilisant le code obtenu **(Étape 4)**.

La requête HTTP ressemblerait à :

```
POST /oauth2/v4/token HTTP/1.1
Host: www.googleapis.com
Content-Type: application/x-www-form-urlencoded

code=code_autorisation&
client_id=id_1&
client_secret=secret_1&
redirect_uri=https://example.com/oauth2-redirect&
grant_type=authorization_code
```

En réponse à cette requête, Google génère un *token* d'accès **(Étape 5)** :

```
{
  "access_token":"access_token_1",
  "expires_in":3000,
  "token_type":"Bearer"
}
```

Listing 1 – Exemple de code d'accès généré par google

Le *token* d'accès a une durée de validité de 50 minutes (3000 secondes). La durée de validité est à l'appréciation du serveur d'autorisation (ici Google) qui fournit une documentation pour les clients qui donne tous les détails nécessaires.
Notre site peut maintenant accéder aux mails de l'utilisateur en rajoutant dans chacune des requêtes une entête HTTP `Authorization` contenant la valeur `Bearer access_token_1`.

i Pour des soucis de rigueur, l'exemple ici contient des détails d'implémentation liés à OAuth 2.0 qui propose un ensemble de contraintes techniques qui régissent les interactions entre les différents acteurs en jeu. L'objectif n'est pas de les comprendre en détails mais de voir d'une manière globale le fonctionnement du mode d'autorisation avec un code.

5.2. Autorisation implicite : *Implicit Grant*

Ce processus est idéal pour les applications clientes dites *publiques*. Comme pour l'autorisation avec un code, il fait intervenir le client, le propriétaire de la ressource protégée et le serveur d'autorisation.
Les clients **publics**, à l'opposé des clients confidentiels, ne peuvent pas assurer la confidentialité de leurs identifiants.

5.2.1. Cas d'usage

Les applications mobiles ou encore les applications JavaScript exécutées sur le navigateur de l'utilisateur sont les principales utilisatrices de ce moyen d'autorisation. Le client étant publique, il ne peut pas s'authentifier de manière fiable.
Ainsi, contrairement à l'autorisation avec un code décrite plus haut, le client ne peut fournir de mot de passe. Son identifiant et l'URL de redirection ***préenregistrée*** permettront de

l'authentifier d'où le nom d'autorisation implicite. Du moment où le serveur d'autorisation redirige le propriétaire de la ressource vers une URL associée au client, il considère implicitement que le client a fait la demande d'autorisation et va traiter la réponse.

5.2.2. Description

Le processus d'autorisation se rapproche beaucoup du précédent avec les étapes d'authentification du client en moins :

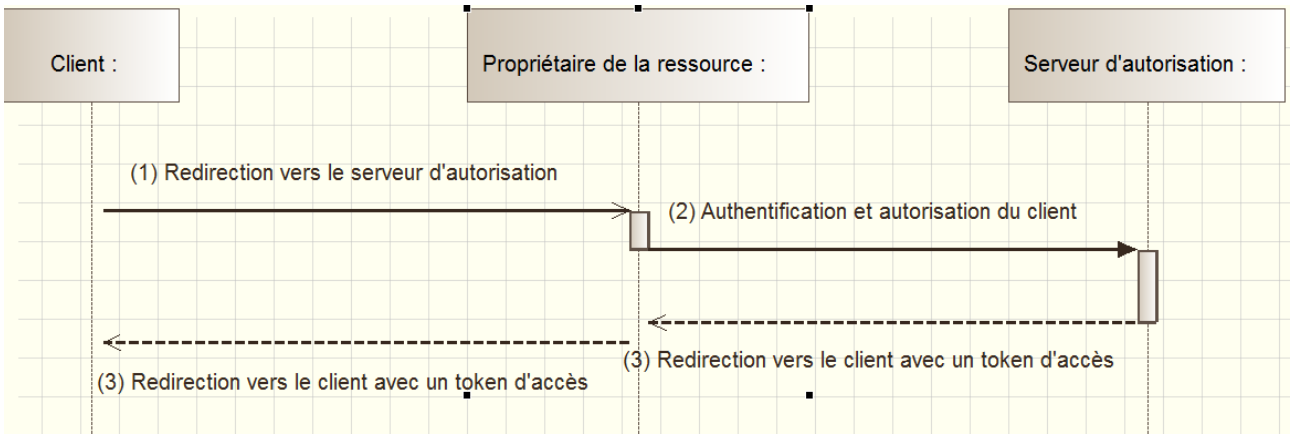

FIGURE 5. – Implicit Grant

1. Le client commence par rediriger le propriétaire de la ressource vers le serveur d'autorisation. Le client doit inclure son identifiant dans la requête de redirection et le niveau d'accès qu'il souhaite obtenir.
2. Le propriétaire de la ressource s'authentifie auprès du serveur d'autorisation et approuve ou non la requête du client.
3. Si la requête est autorisée, le serveur d'autorisation redirige à nouveau le propriétaire de la ressource en utilisant l'URL de redirection défini par le client. La requête de redirection contient dans son URL le *token* d'accès qui permettra d'accéder aux ressources protégées.

5.2.3. Exemple d'utilisation

Un exemple courant pour ce type d'autorisation est l'utilisation de la fonctionnalité Facebook Login .
Considérons que nous sommes en train de développer une application mobile Windows 8 et pour s'authentifier, nous demandons aux utilisateurs de nous donner accès à leurs comptes Facebook .
Pour les applications clientes en JavaScript, Facebook propose un SDK pour implémenter cette fonctionnalité, mais pour notre cas, nous allons utiliser la méthode manuelle .
Note site doit être déclaré sur Facebook avec un identifiant et une URL de redirection. Pour cet exemple, nous prendrons comme identifiant `id_1` et comme URL de redirection `ms-app://SID_de_notre_application`.

i

Sous Windows 8, les URL du type *ms-app ://appSID* redirige vers l'application mobile ayant le SID spécifié dans l'URL. Le SID (**s**ecurity **id**entifier) est un identifiant de sécurité

unique pour notre application.

Facebook comme tout serveur d'autorisation expose des endpoints pour interagir avec lui :

- https ://www.facebook.com/dialog/oauth pour initialiser le processus d'autorisation ;
- et https ://www.facebook.com/oauth/access_token pour obtenir un *token* d'accès et éventuellement un *token* de rafraîchissement.

Comme spécifié dans la description de ce processus d'autorisation, notre application mobile commence par rediriger l'utilisateur vers l'URL https ://www.facebook.com/dialog/oauth ?client_id=id_1&display=popup&redirect_uri=ms-app ://SID_de_notre_application&response_type=token **(Étape 1)**.
Les paramètres utilisés dans cet exemple sont identiques à ceux utilisés dans le premier. Seul le paramètre `response_type` passe de `code` à `token`.
Cette redirection affiche la boîte de dialogue Login de Facebook.
Si l'utilisateur s'authentifie sur Facebook (ou est déjà authentifié) et autorise notre application à accéder à ses informations **(Étape 2)**, Facebook le redirige vers l'URL ms-app ://SID_de_notre_application avec un *token* d'accès prêt à l'emploie **(Étape 3)**.
Vous remarquerez que l'URL https ://www.facebook.com/oauth/access_token n'est pas utiliser dans ce processus car le serveur d'autorisation renvoie directement un *token* d'accès.

5.3. Autorisation avec les identifiants du propriétaire de la ressource : *Resource Owner Password Credentials Grant*

Ce processus se démarque des deux premières par le fait qu'il ne nécessite pas de redirection du propriétaire de la ressource vers le serveur d'autorisation

5.3.1. Cas d'usage

Le seul cas d'usage valide pour cette méthode est lorsque le client et le serveur d'application sont fournis par la même entité (même entreprise, même développeur, etc.).
Il doit y avoir une relation de confiance absolue entre ces deux acteurs car le propriétaire de la ressource doit fournir ses identifiants au client.
Comme décrit dans la partie « La naissance de OAuth », fournir ses identifiants à des applications tierces est très fortement déconseillé et OAuth a été pensé pour éviter cela.

Cette méthode ne doit donc être utilisée que **si et seulement si** le serveur d'autorisation et le client sont issus d'une même entité.

5.3.2. Description

Le processus d'autorisation est assez simple :

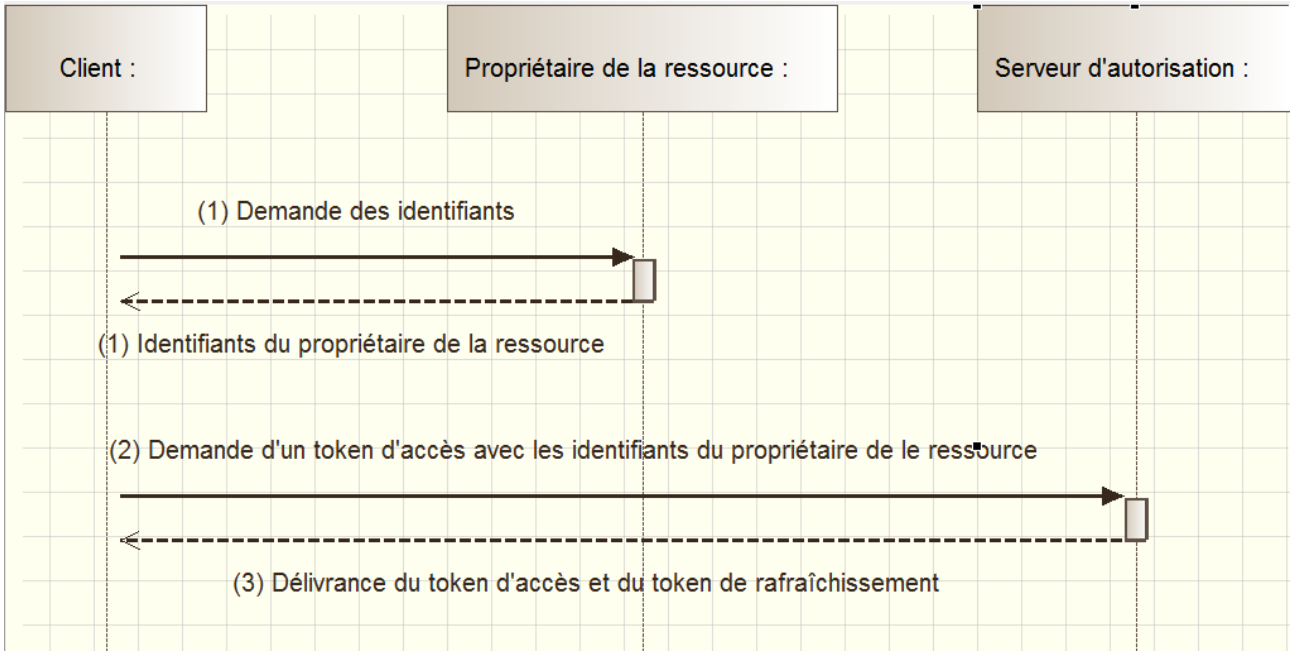

FIGURE 5. – Resource Owner Password Credentials Grant

1. Le client commence par demander au propriétaire de la ressource ses identifiants.
2. Le client demande ensuite un *token* d'accès en utilisant les identifiants du client mais **en rajoutant aussi ses propres identifiants** afin de s'authentifier auprès du serveur d'autorisation.
3. Une fois le client authentifié par le serveur d'authentification, le serveur valide les identifiants du propriétaire de la ressource et génère, le cas échéant, un *token* d'accès et de rafraîchissement.

Il est utile de préciser qu'à l'étape 3, le serveur d'autorisation doit aussi s'assurer que le client a bien le droit d'utiliser ce type d'authentification.

5.3.3. Exemple d'utilisation

Comme déjà énoncé plus haut, cette méthode doit être utilisée avec parcimonie. Mais il existe bel et bien des cas où son utilisation reste pratique.
Prenons encore l'exemple de Google. Pour accéder à Gmail depuis un site web, nous devons nous connecter.
Google étant lui-même le serveur d'autorisation et l'éditeur du site web et de l'application mobile, l'utilisation des identifiants du propriétaire de la ressource n'est pas problématique.
De manière générale, si nous sommes propriétaires du serveur d'autorisation et du client, cette méthode peut être plus simple et rapide car l'utilisateur ne fournit pas ses identifiants à une application tierce : l'objectif de base de OAuth 2.0 est toujours respecté.
Donc si nous proposons une API avec une sécurité basée sur OAuth 2.0, nous pouvons utiliser ce mode d'autorisation pour permettre à notre propre application cliente d'accéder aux ressources protégées.
Reprenons notre client *example.com* avec comme identifiant `id_1`, comme mot de passe `secret_1`.
Supposons que le serveur d'autorisation propose un *endpoint* oauth2.example.com/token pour obtenir des *tokens* d'accès.

Ainsi, nous pouvons présenter un formulaire à l'utilisateur pour obtenir ces identifiants. Lorsqu'il valide le formulaire, nous récupérons ces informations **(Étape 1)**.
Avec ces identifiants, le serveur peut faire une requête HTTP de ce type pour obtenir un *token* d'accès **(Étape 2)** :

```
POST /token HTTP/1.1
Host: oauth2.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=johndoe&password=password&client_id=id_1&client_se
```

La réponse du serveur d'autorisation *oauth2.example.com* à une telle requête sera un *token* d'accès **(Étape 3)**.

5.4. Autorisation avec les identifiants du client : *Client Credentials Grant*

Ce processus est le seul qui ne fait intervenir que deux acteurs :
- le client ;
- et le serveur d'autorisation.

Il est souvent appelé two-legged OAuth (deux jambes) ou 2LO.
Le propriétaire de la ressource n'est pas sollicité car cette méthode ne permet pas en général d'accéder à des ressources protégées (à part ceux du client).

5.4.1. Cas d'usage

L'utilisation de ce mode d'autorisation peut s'avérer très utile si un serveur fournit une API publique mais veut ajouter une limitation à l'usage qui est faite par les clients.
Ainsi, le serveur pourra mettre en place par exemple un système de limitation du nombre de requêtes par client ou encore permettre à un client de configurer ses propres paramètres.

5.4.2. Description

Le processus d'autorisation se réduit à deux étapes :

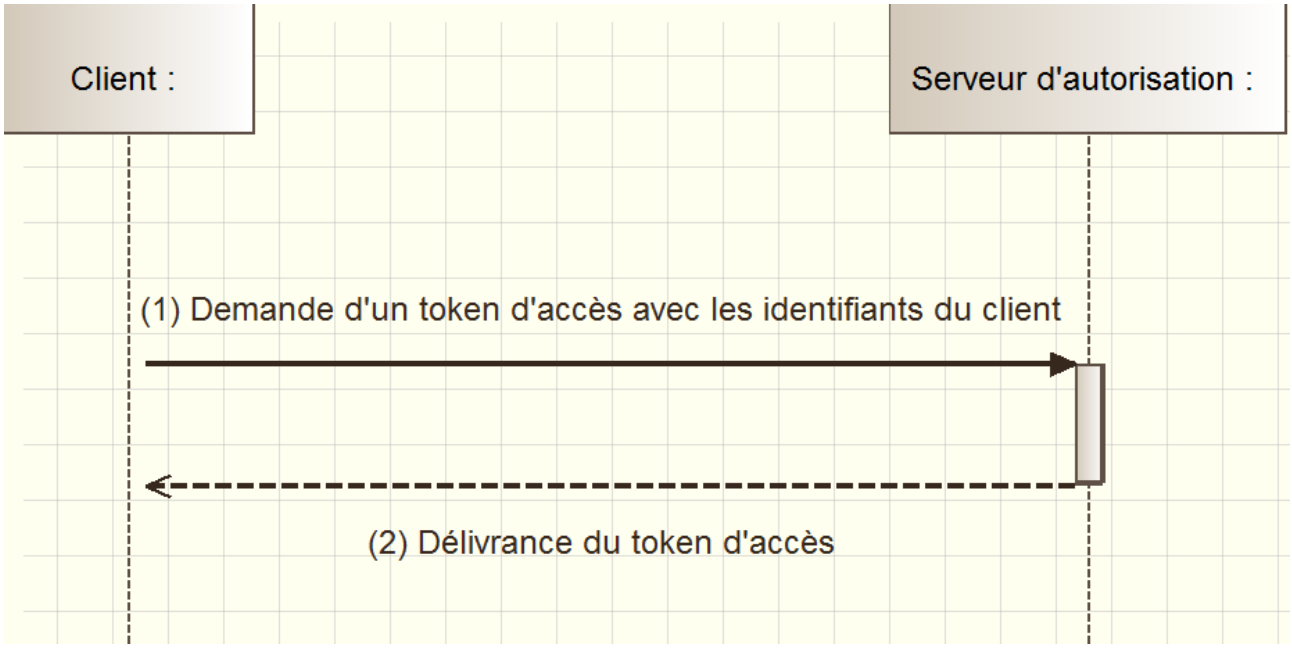

FIGURE 5. – Client Credentials Grant

1. Le client envoie ses identifiants au serveur d'autorisation pour obtenir un *token* d'accès.
2. Si les identifiants du client sont valides, le serveur d'autorisation génère un *token* pour ce client.

5.4.3. Exemple d'utilisation

Nous allons rester avec Facebook, avec sa notion de token d'accès d'app .
Ce type de *token* d'accès est nécessaire pour modifier et lire les paramètres d'une application Facebook.
Pour en obtenir une, l'application doit faire une requête sur l'URL de génération de *token* de Facebook **(Étape 1)**.
Si nous avions une application Facebook avec comme identifiant 'id_1' et comme mot de passe 'secret_1', la requête ressemblerait à :

```
GET /oauth/access_token
    ?client_id=id_1
    &client_secret=secret_1
    &grant_type=client_credentials
```

Si tout se passe bien, le serveur d'autorisation renverrait une réponse contenant notre *token* d'accès **(Étape 2)**.

```
 {
 "access_token": "<access-token>",
 "token_type": "<type>",
 "expires_in": "<seconds-til-expiration>"
}
```

Dans les scénarios où le client est considéré comme confidentiel et doit donc s'authentifier (*Authorization Code Grant*, *Resource Owner Password Credentials Grant* et *Client Credentials Grant*), le *token* d'accès obtenu est confidentiel dans le sens où seul le client et les serveurs d'autorisation et de ressource y ont accès. Tandis que pour l'autorisation implicite (*Implicit Grant*), le *token* d'accès est exposé au propriétaire de la ressource (ce *token* est présent dans l'URL de redirection qui est accessible).

OAuth 2.0 propose un large panel permettant de mettre en place un système d'autorisation standardisé.
Tous les différents scénarios mis en œuvre permettent, chacun avec son approche, d'obtenir un token d'accès et répondent à un besoin simple : accéder de manière sécurisée à une ressource protégée au nom de son propriétaire.
Les documentations officielles de Google et de Facebook contiennent des exemples simples sur l'utilisation du framework OAuth 2.0 et sur les possibilités d'extension de celui-ci.
Il existe, par ailleurs, un ensemble de risques connus qui pourrait rendre une implémentation de OAuth 2.0 vulnérable.
Les règles de sécurité en jeu pour implémenter un serveur OAuth 2.0 fiable étant assez nombreuses et complexes, il est donc préférable d'utiliser une librairie déjà existante pour faciliter le travail si votre objectif est juste d'utiliser OAuth 2.0.
Pour créer soi-même un serveur OAuth 2.0, la RFC 6749 reste la meilleure référence.