#06
Hashs & Tokens

Une API Rest est Stateless

Une API Rest est un protocole ***"stateless" (sans état).***

Chaque communication entre un client et l'API est indépendante. **L'API ne persiste pas d'état entre 2 requêtes successives**. Seules les valeurs stockées dans la base de données persistent.

Il revient au client de fournir à l'API les informations utiles (clé d'API, id de ressource, token...) dans chaque requête pour que le serveur re-créé le *"contexte"*.

Chaque requête à l'API doit être auto-suffisante.

Scénario *"Stateful"*

A l'inverse, un scénario "Stateful" signifie que le contexte est préservé par l'API d'un échange client / serveur à l'autre.

Ce type de scénario est généralement rendu possible par l'ouverture d'une session et la persistance du contexte à travers des variables de session.

`POST /orders`	---------->	création d'une commande (création du contexte dans la session et/ou écriture en base de données)
`POST /orders/add-items`	---------->	ajout d'items à la commande (contexte contenu dans la session)
`POST /orders/payment`	---------->	finalisation de la commande

L'exemple ci-dessus représente un scénario "Stateful" dans lequel plusieurs requêtes se succèdent avec un contexte maintenu par une session.

Scénario *"Stateless"*

Un scénario "Stateless" implique que le client fournisse à chaque requête une information (ex : uuid, hash, token...) utile à l'obtention du contexte (ex : stocké en base de données).

`POST /orders`	---------->	création d'une commande en base de données
	<---------	transmission au client de l'identifiant unique associé à la commande (ex : 123).
`POST /orders/123/items`	---->	transmission au serveur de l'identifiant unique associé à la commande, pour recréer le contexte (ex : lecture dans la base de données) et ajouter des items à cette commande.

Le caractère *"Stateless"* d'une API offre plusieurs avantages :

- ***"Load Balancing"*** : capacité à répartir la charge du back end sur plusieurs instances de serveurs autonomes,
- capacité à gérer plusieurs appels d'API parallèles,
- capacité à utiliser un système de cache pour accélérer le traitement des requêtes GET,

Comment contextualiser l'API ?

Fournir un identifiant unique

La transmission des données entre le client et le serveur peut représenter un risque de sécurité selon la nature et la criticité des informations communiquées.

Pour identifier une ressource dans une base de données, il est nécessaire de connaître l'identifiant unique associé à cette ressource.

Généralement l'identifiant unique correspond à un nombre entier auto-incrémenté (SQL) ou une chaîne de caractères alphanumériques (NoSQL) générés automatiquement par la base de données.

Pour éviter de révéler la logique interne (ex: prochain id de commande) et le type de base de données, il est possible de recourir à l'emploi d'un *"hash" ou d'un "identifiant opaque" (type UUID)* généré par le serveur, faisant office d'identifiant unique.

Le hash n'étant pas intelligible, les utilisateurs ne peuvent pas deviner le type de base de données utilisé ni la logique de création d'identifiant unique.

Plusieurs solutions permettent de générer un identifiant unique opaque :

- **UUID** (pour disposer d'une chaîne normée),
- ou un **hash crypté** (pour disposer d'une chaîne complexe non normée).

UUID

*"**UUID**"* signifie *"Universal Unique IDentifier"*.

L'unicité (mondiale) d'un UUID est *"hautement probable"* mais pas garantie.

Il y a 5×10^{38} combinaisons possibles.

Il existe plusieurs versions de UUID, chaque version renforçant l'unicité probable des codes générés (dernière version : 5).

Un UUID se présente sous la forme d'un groupe de caractères hexadécimaux, en minuscules, séparés par des tirets :

```
85368d10-41f1-11ea-a91f-b3cfc7d5a712
```

La valeur des UUID est codée sur 128 bits, est obtenue à partir d'algorithmes et d'après les caractéristiques physiques de l'ordinateur qui les génère (ex : adresse MAC...).

UUID avec JS

Plusieurs modules NPM permettent d'obtenir des UUID dans un script JS :

https://www.npmjs.com/package/uuid

Installation du module NPM **uuid** :

```
npm i uuid
```
ou
```
yarn add uuid
```

Utilisation du module npm uuid au sein d'une application Node.js:

```
const uuidv1 = require('uuid/v1');
const uuid = uuidv1();
```

Token

Token signifie *"jeton"*.

Un token se présente sous la forme d'une longue chaîne de caractères alphanumériques, générée par le serveur.

Il reprend le principe de l'UUID afin d'identifier de façon unique une ressource (utilisateur, collection de données associée à un utilisateur…) dans un système *"stateless"*.

La transmission d'un token en paramètre d'une route d'API permet de re-créer un *"contexte"* dans un système *"stateless",* afin d'authentifier ou d'identifier le client (vérifier son droit d'accès à l'API et/ou obtenir des informations spécifiques, associées au client).

Dans un système stateless, l'utilisation d'un token se substitue à l'utilisation d'un cookie de session.

Un token peut être communiqué à l'API de façon *"opaque"* (visible) ou *"transparente"* (invisible).

Hashs chiffrés

Pour renforcer la sécurité du système, les tokens peuvent être générés sous forme de chaîne de caractères alphanumériques complexe et cryptés.

Lorsqu'un hash crypté est communiqué par le client, il est possible de d'abord vérifier son authenticité avant de vérifier s'il est associé à une ressource dans la base de données.

L'intérêt est d'éviter de consulter la base de données dans le cas où le hash n'est pas authentique, ce qui représente une économie de charge liée à une opération I/O.

Génération de hash crypté avec Bcrypt

Le module NPM **Bcryptjs** permet de générer des tokens complexes et cryptés.

https://github.com/dcodeIO/bcrypt.js#readme

Génération synchrone de hash avec Brcypt côté serveur, avec Node.js :

```
const bcrypt = require('bcryptjs');
const salt = bcrypt.genSaltSync(10);
const hash = bcrypt.hashSync("B4c0/\/", salt);
```

Génération automatique de hash et salt

```
const hash = bcrypt.hashSync('bacon', 8);
```

Vérification synchrone de l'authenticité d'un token côté serveur avec Node.js

```
bcrypt.compareSync("B4c0/\/", hash); // true
bcrypt.compareSync("not_bacon", hash); // false
```

Vérification asynchrone de l'authenticité d'un token côté serveur avec Node.js

```
bcrypt.compare("B4c0/\/", hash).then((res) => {
    // res === true
});
```

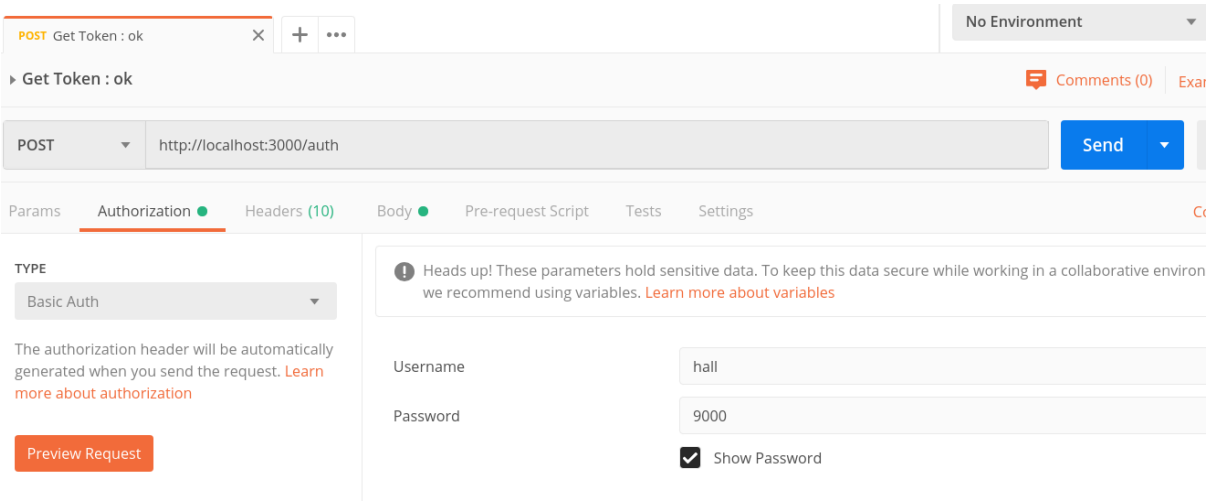
Authentification

Pour authentifier un utilisateur auprès d'une API, 2 méthodes alternatives sont fréquemment utilisées :

- via ***OAuth***,
  ou
- via ***JWT*** (*"JSON Web Token"*).

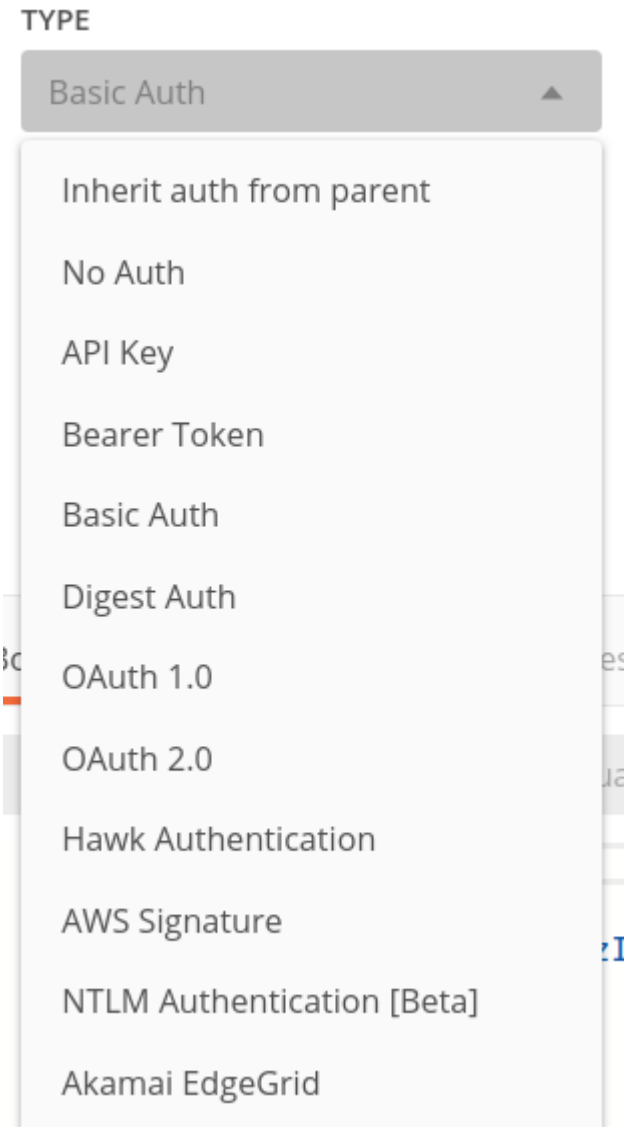
Authentification avec Basic Auth

L'authentification de type *"Basic Auth"* consiste à transmettre les identifiants (aussi appelés "Credentials") au serveur. Ces identifiants sont automatiquement cryptés en base64 lors de l'envoi.
Les identifiants doivent être décryptés avant d'être vérifiés (généralement en comparaison avec les enregistrements en base de données).



Plusieurs modes d'authentification sont possibles via le header *"Authorization"* HTTP :

- Basic Auth
- Bearer Token
- OAuth
- …



L'avantage est de ne pas communiquer de façon transparente les credentials (login / password) dans la requête HTTP.

Une authentification via ces différentes méthodes fournit les credentials cryptés (Base64).

Documentation Postman :
https://learning.postman.com/docs/postman/sending-api-requests/authorization/

Authentification Basic Auth avec Node.js

```
const base64Credentials = req.headers.authorization.split(' ')[1];
const credentials = Buffer.from(base64Credentials, 'base64').toString('ascii');
const [login, password] = credentials.split(':');
```

Le script accède aux credentials via le header "*Authorization*", les déconcatène les décrypte.

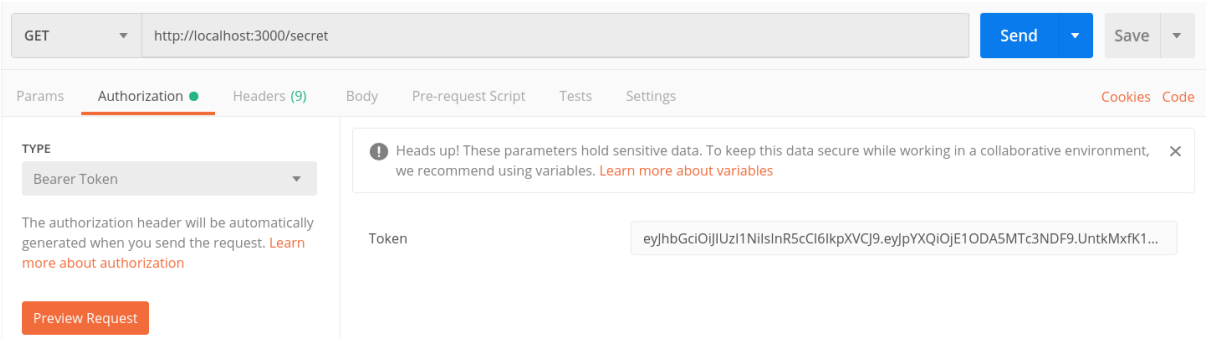
Il est ensuite nécessaire de vérifier que le login existe dans la base de données et que le mot de passe est correct (dans l'exemple ci-dessous, la vérification de l'utilisateur est basique).

```
app.post('/auth', (req, res) => {


    let login, password;

    //authentification Basic Auth avec Login et Mdp fournis dans le header Authorization
    if (req.headers.authorization) {
        const base64Credentials = req.headers.authorization.split(' ')[1];
        const credentials = Buffer.from(base64Credentials, 'base64').toString('ascii');
        [login, password] = credentials.split(':');


    } else {
        return res.status(401).end("Missing credential");
    }

    const user = users.find(u => u.login === login && u.password === password);

    if (!user)
        return res.status(401).end("Bad credentials");

    res.setHeader("Content-Type", "application/json");
    let privateKey = fs.readFileSync('./jwt_secret.txt', 'utf8');

    let token = jwt.sign({}, privateKey, { algorithm: 'HS256' });
    res.send({ token: token });
});
```

Vérification de token via Authorization Bearer Token

```
const fs = require("fs");
const jwt = require('jsonwebtoken');


module.exports = function isAuthorized(req, res, next) {

    if (typeof req.headers.authorization !== "undefined") {

        let token = req.headers.authorization.split(' ')[1];//Bearer Authorization
        let privateKey = fs.readFileSync('./jwt_secret.txt', 'utf8');

        jwt.verify(token, privateKey, { algorithm: "HS256" }, (err, user) => {
            if (err) {
                res.status(500).json({ error: "Not Authorized" });
                throw new Error("Not Authorized");
            }
            return next();
        });
    } else {
        res.status(500).json({ error: "Not Authorized" });
        throw new Error("Not Authorized");
    }
}
```

Middleware utilisé dans Express.js permettant de vérifier à chaque requête si le Token JWT fourni via le header Authorization (en mode Bearer Token) est correct.



Transmission du token JWT via le header Authorization en mode Bearer Token.

Le token est transmis au serveur via le header Authorization en mode Bearer Token, il doit ensuite être vérifié par le serveur.

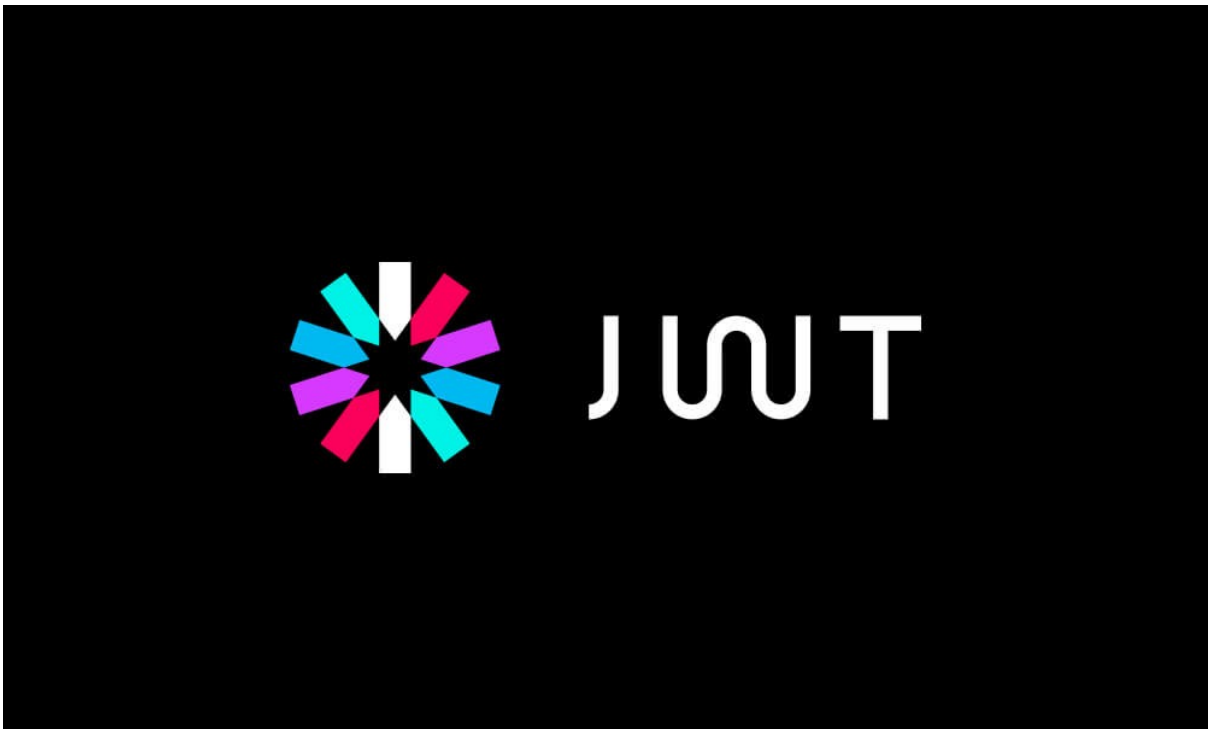
Authentification par "Capability" (opaque)

Un token est ***"opaque"*** lorsqu'il est **communiqué de façon visible** dans l'URL ou dans le Header d'une requête HTTP.

Un token classique nécessite côté serveur, l'établissement d'un registre (table de données, fichier...) où chaque token est associé à un utilisateur ou une ressource. Ce registre permet de vérifier l'authenticité du token. L'inconvénient est que cela implique de consulter la base de données ou le fichier très souvent.

La mise en oeuvre d'un token opaque est simple mais moins sécurisée qu'un token *"transparent"* (invisible) : il suffit que l'URL soit partagée accidentellement (ou non) pour que les données soient accessibles par n'importe quel utilisateur sans contrainte.

Authentification avec JWT (transparente)



Le token JWT (https://jwt.io/) **a l'avantage d'être transparent (invisible) et à durée limitée, donc plus sécurisé qu'un token classique.**

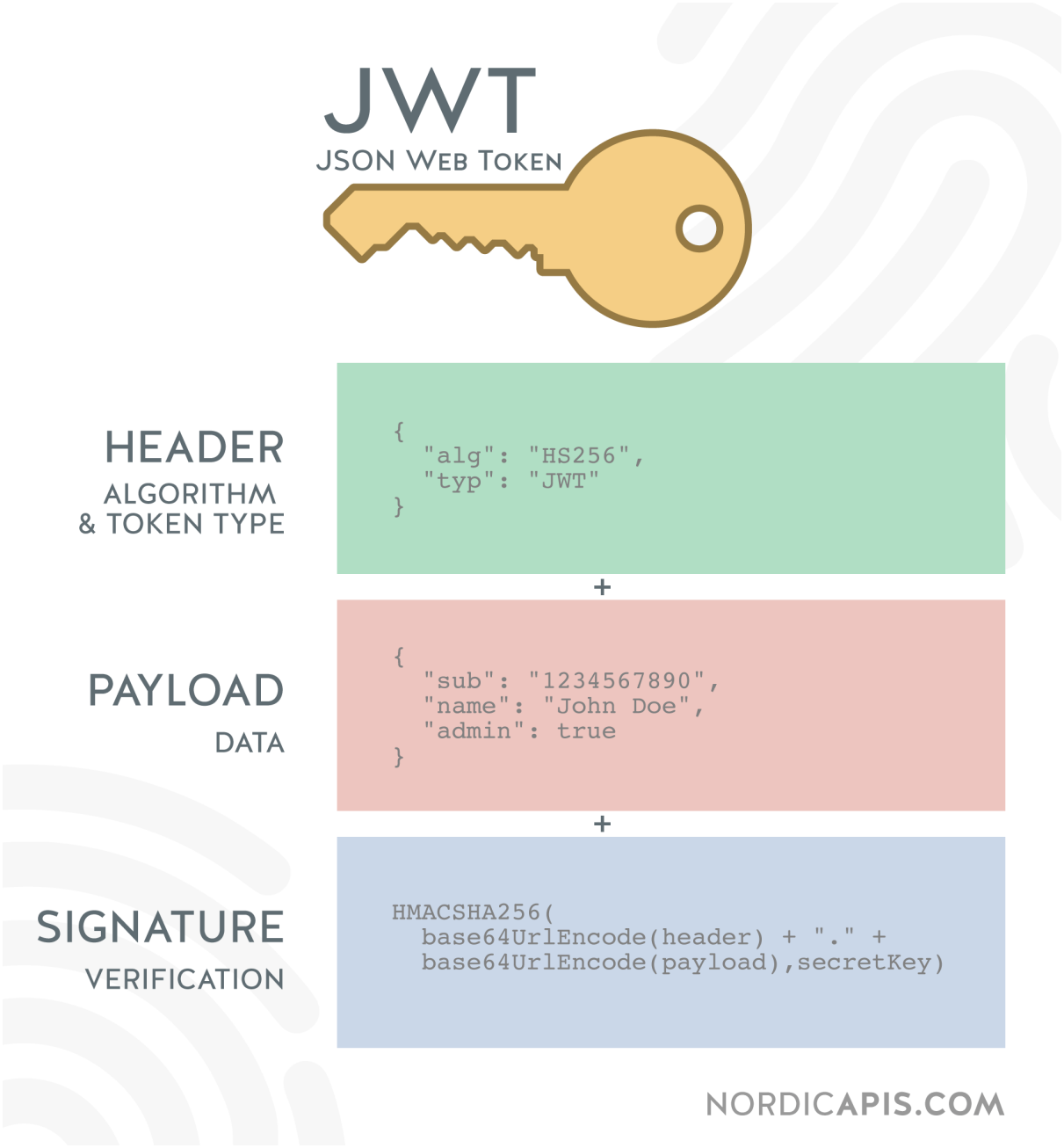
Un token JWT est généré et signé par le serveur, qui est **la seule entité capable de vérifier l'authenticité de la signature du token via un *"secret"*.**

La vérification d'authenticité d'un token JWT s'effectue sans **consultation de la base de données (ou autre type de registre) comme c'est le cas pour un token opaque, ce qui représente une économie de charge**. La vérification nécessite simplement la consultation des claims du *"Payload"*.

JWT présente l'intérêt de ne pas nécessiter l'interrogation d'une autorité d'authentification externe (comme c'est le cas avec *OAuth*) pour vérifier l'authenticité d'un token et obtenir les informations liées.

JWT est un standard ouvert (RFC 7519) qui se compose de 3 chaînes de caractères séparées par un point, correspondant à :

- un header (entête),
- un payload (contenu),
- une signature



Le header et le payload sont structurés en JSON et encodés en base64Url.

Le header permet d'identifier l'algorithme utilisé pour générer la signature et le type de token.

Header indiquant que l'algorithme utilisé est *HMAC-SHA256* (HMAC ou RSA + SHA512, BCRYPT...) et que le type de token est JWT.

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

Le payload contient les informations destinées à être transmises entre le client et l'API. Un payload contient plusieurs *"claims"* (clés) dont certaines sont prédéfinies et standardisées, et d'autres sont ajoutées par le fournisseur de l'API.

Claims du payload :

```
{
     "sub": "Edward Snowden",
     "exp": "1545926973",
     "aud": "true",
     "iss":
}
```

Claim obligatoire prédéfini :

- exp (Expiration Time) : correspond à la date d'expiration du token,

Claims optionnels prédéfinis :

- sub (Subject) : sujet du token (utilisateur identifié par le token),
- aud (Audience) : destinataire du token (permet de vérifier que le destinataire du token, soit le serveur cible, est le bon),
- iss (Issuer) : émetteur du token,
- iat (Issued at) : date d'émission du token,
- nbf (Not Before) : date de début de validité du token,

https://tools.ietf.org/html/rfc7519#section-4.1

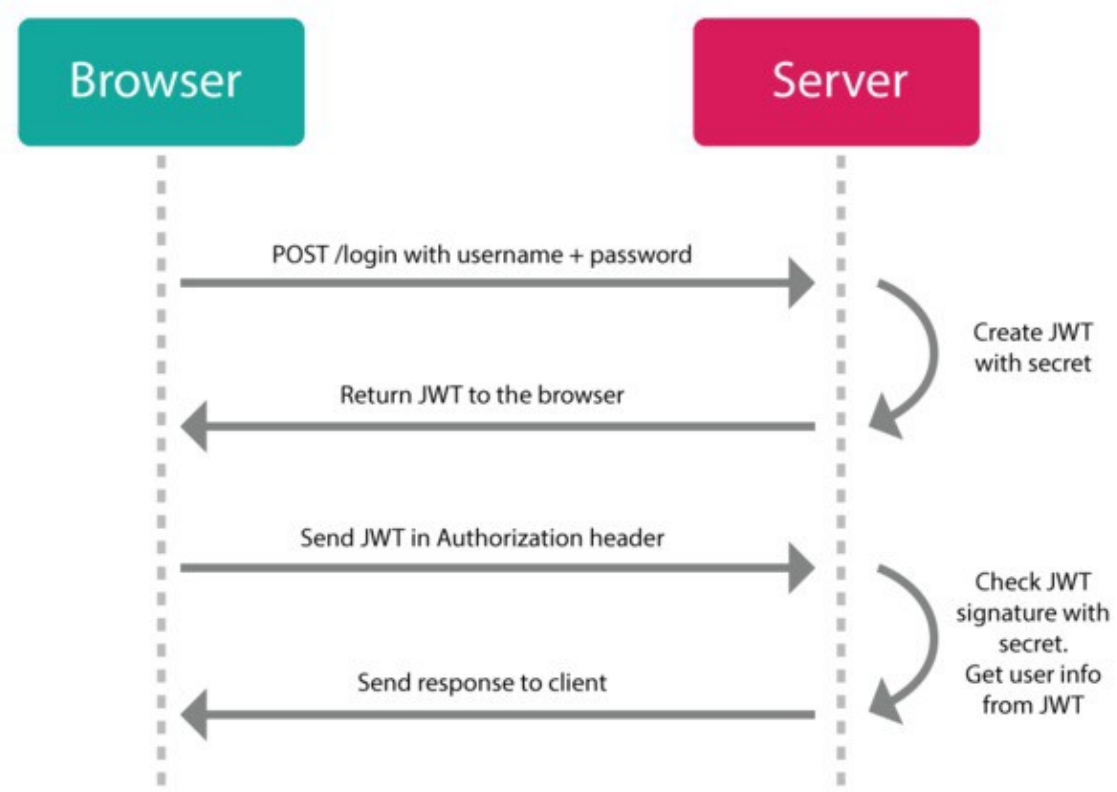
La signature résulte du header et du payload. Si la signature est invalide, le token est rejeté.

Dans le cadre d'une authentification, le token associé à un utilisateur a une durée de validité définie par l'API.

JWT en pratique

Un jeton JWT est communiqué par le serveur au client après que le client se soit authentifié de façon traditionnelle (ex : requête POST contenant les *"credentials"* dans le body : identifiant / mot de passe) ou ait initié un événement spécifique (ex : initiation d'une commande ou d'une procédure en plusieurs étapes).

Généralement, le JWT est fourni au client après que ce-dernier ait effectué une requête de type POST à une URI *"/auth"* ou *"/login"* de l'API.



Plutôt que de faire transiter son identifiant et son mot de passe à chaque requête, le client communique ensuite le token en tant que paramètre (dans l'url ou dans les headers).

Génération et vérification d'un token JWT

Pour s'assurer de l'authenticité d'un token, le serveur doit vérifier :

- que les informations contenues dans le token (header / body / signature) sont cohérentes,
- que la date de validité du token est acceptable,

```
token=base64url(head)+"."+baser64url(body)+"."+signature
```

Le module NPM **JsonWebToken** se charge de générer et vérifier l'authenticité des tokens.

https://www.npmjs.com/package/jsonwebtoken

Génération synchrone d'un token à partir d'une clé secrète stockée sur le serveur dans un fichier nommé *"private.key"* (nom du fichier et contenu libres) :

```
const privateKey = fs.readFileSync('private.key');
const token =
jwt.sign(
{ foo: 'bar' },
privateKey,
{ algorithm: 'RS256'});
```

Génération synchrone d'un token avec une durée de validité de 1h (= 3600 secondes):

```
jwt.sign(
{  data: 'foobar'},
'secret',
{ expiresIn: '1h' }
);
```

```
jwt.sign({
  exp: Math.floor(Date.now() / 1000) + (60 * 60),
  data: 'foobar'
}, 'secret');
```

Avec **Express.js**, il est pratique de mettre en place un ***middleware*** qui s'exécutera à chaque appel de l'API pour vérifier que le token transmis par le client est correct, avant de gérer la route.

Attention cependant à ne pas exécuter dans la requête initiale (ex : /auth ou /login) permettant de générer le token.

Exemple de ***middleware d'authentification*** permettant de vérifier l'authenticité d'un token généré par JWT :

```
const fs = require("fs");
const jwt = require('jsonwebtoken');


module.exports = function isAuthorized(req, res, next) {
    if (typeof req.headers.authorization !== "undefined") {
        // retrieve the authorization header and parse out the
        // JWT using the split function
        let token = req.headers.authorization;
        let privateKey = fs.readFileSync('./jwt_secret.txt', 'utf8');
        // Here we validate that the JSON Web Token is valid and has been
        // created using the same private pass phrase
        jwt.verify(token, privateKey, { algorithm: "HS256" }, (err, user) => {

            // if there has been an error...
            if (err) {
                // shut them out!
                res.status(500).json({ error: "Not Authorized" });
                throw new Error("Not Authorized");
            }
            // if the JWT is valid, allow them to hit
            // the intended endpoint
            return next();
        });
    } else {
        // No authorization header exists on the incoming
        // request, return not authorized and throw a new error
        res.status(500).json({ error: "Not Authorized" });
        throw new Error("Not Authorized");
    }
}
```

Exemples d'utilisation de JWT :

https://www.grafikart.fr/tutoriels/json-web-token-presentation-958

https://medium.com/swlh/a-practical-guide-for-jwt-authentication-using-nodejs-and-express-d48369e7e6d4

https://www.sohamkamani.com/blog/javascript/2019-03-29-node-jwt-authentication/

https://github.com/sohamkamani/jwt-nodejs-example

https://marmelab.com/blog/2020/07/02/manage-your-jwt-react-admin-authentication-in-memory.html

A noter :

Postman permet de renseigner des variables utilisables au sein des requêtes, ce qui facilite par exemple la mise à jour du token dans toutes les requêtes nécessitant une authentification. Les variables peuvent aussi être utilisées pour modifier facilement des données communes à toute l'API (ex le port ou le nom de domaine visé).

https://learning.postman.com/docs/postman/variables-and-environments/variables/