JWT

la clé de voûte des échanges sécurisés entre vos services

Comprendre, adopter et sécuriser le JSON Web Token dans une architecture moderne

Livre blanc by ChallengeMyProject

À qui s'adresse ce document ?

Aux décideurs techniques qui veulent comprendre pourquoi le JWT s'est imposé comme un standard, et aux développeurs qui veulent l'implémenter correctement. Les deux profils y trouveront de quoi décider, puis de quoi construire.

01

Le problème : authentifier sans ralentir

Pourquoi le modèle de session classique ne suit plus ?

Dès qu'une application dépasse le stade du prototype, une question revient systématiquement : comment savoir, à chaque requête, qui parle et ce qu'il a le droit de faire ?

Pendant longtemps, la réponse tenait en un mot : **la session**. À la connexion, le serveur créait une session en mémoire ou en base, et renvoyait au navigateur un identifiant sous forme de cookie. À chaque requête suivante, le serveur retrouvait la session correspondante et savait à qui il avait affaire.

Ce modèle fonctionne très bien tant que l'on a un seul serveur. Mais les architectures actuelles sont rarement aussi simples. On a un front découplé du back, plusieurs instances derrière un load balancer, des microservices qui s'appellent entre eux, des applications mobiles, des intégrations tierces.

Dans ce contexte, stocker l'état d'authentification au même endroit que celui qui le consulte devient un goulot d'étranglement : il faut soit partager la mémoire de session entre toutes les instances, soit interroger une base à chaque appel.

Le JSON Web Token (JWT) propose un changement de paradigme : et si le jeton lui-même contenait toute l'information nécessaire, de manière vérifiable, sans que le serveur ait besoin de consulter quoi que ce soit ? C'est exactement ce que permet le JWT, et c'est ce qui explique son adoption massive.

02

Anatomie d'un JWT

Trois blocs, une signature, aucun secret bien gardé

Un JWT est une chaîne de caractères composée de trois parties séparées par des points. Chaque partie est encodée en *Base64url* (un encodage, **pas un chiffrement**).

`eyJhbGciOiJ…` . `eyJzdWIiOi…` . `SflKxwRJSME…`

en-tête — charge utile — signature

EN-TÊTE

Type et algorithme

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

Indique comment le jeton est signé.

CHARGE UTILE

Les claims (affirmations)

```
{
  "sub": "1234",
  "role": "admin",
  "aud": "api…",
  "exp": 1700003600
}
```

Lisible par tous — jamais de secret ici.

SIGNATURE

Le sceau de confiance

Calculée avec une clé. Toute modification du jeton la rend invalide.

L'en-tête (*header*) décrit le type de jeton et l'algorithme de signature utilisé.

La charge utile (*payload*) contient les claims, c'est-à-dire les affirmations sur l'utilisateur ou le contexte. On y trouve des champs standardisés et des champs personnalisés.

La signature est ce qui rend le tout digne de confiance. Elle est calculée à partir de l'en-tête, de la charge utile et d'une clé secrète (ou privée). Si quelqu'un modifie ne serait-ce qu'un caractère du payload, la signature ne correspond plus, et le serveur rejette le jeton.

03

Stateless vs stateful : l'intérêt réel du JWT

Ce que vous gagnez quand le serveur arrête de se souvenir

C'est ici que se joue la valeur du JWT, et c'est la partie qui intéresse autant l'architecte que le responsable de la facture cloud.

Dans un modèle **stateful** (à état), le serveur doit se souvenir de chaque utilisateur connecté. Plus il y a d'utilisateurs, plus le store de sessions grossit, et chaque vérification implique un aller-retour vers ce store.

Dans un modèle **stateless** (sans état) reposant sur le JWT, le serveur ne se souvient de rien. Le jeton porte lui-même l'identité et les droits, et le serveur se contente de vérifier la signature — une opération cryptographique locale, rapide, qui ne touche ni la mémoire partagée ni la base. Concrètement, cela se traduit par :

Une scalabilité horizontale naturelle
Vous pouvez ajouter des instances de serveur à volonté : aucune n'a besoin de connaître l'état des autres. Un jeton émis par l'instance A est validé sans problème par l'instance B.

Une réduction de la charge sur la base de données
L'authentification ne génère plus de requête à chaque appel. Sur une API à fort trafic, c'est un gain mesurable.

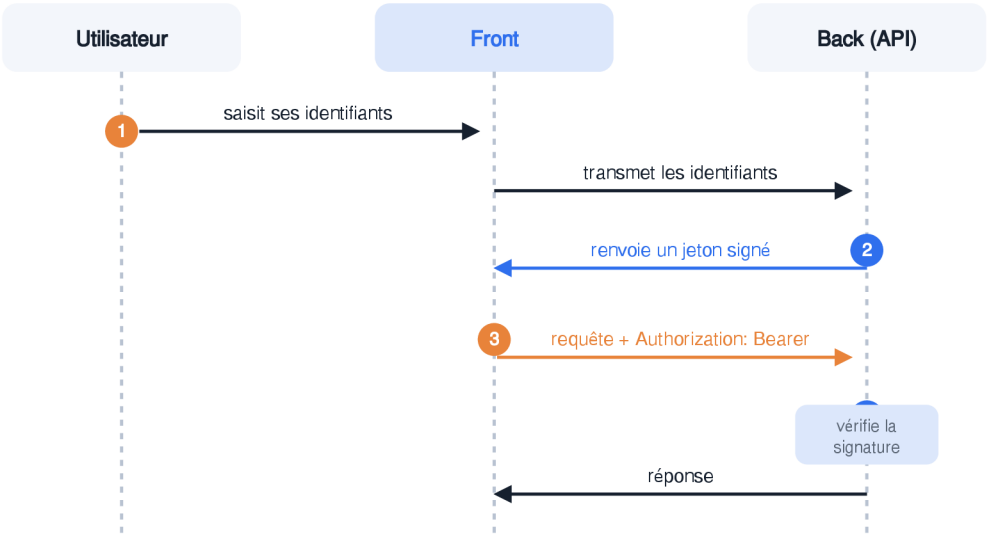
Un découplage propre
Le service qui émet les jetons (l'authentification) peut être totalement séparé des services qui les consomment. C'est la fondation des architectures à API et des microservices..

04

Le JWT dans les échanges front / back

Du login à chaque requête, le parcours d'un jeton

Le cas d'usage le plus courant. Une application web ou mobile (le front) dialogue avec une API (le back). Le déroulé typique est le suivant :



À partir de l'étape 3, aucune consultation externe : le back lit tout dans le jeton.

L'utilisateur saisit ses identifiants. Le back les vérifie, et s'ils sont valides, il génère un jeton signé qu'il renvoie au front. À partir de là, le front joint ce jeton à chacune de ses requêtes, généralement dans l'en-tête *HTTP Authorization*.

À réception, le back vérifie la signature et les claims du jeton. S'ils sont valides, il sait immédiatement qui fait la requête et avec quels droits — sans aucune consultation externe. Si le jeton est absent, expiré ou falsifié, la requête est rejetée.

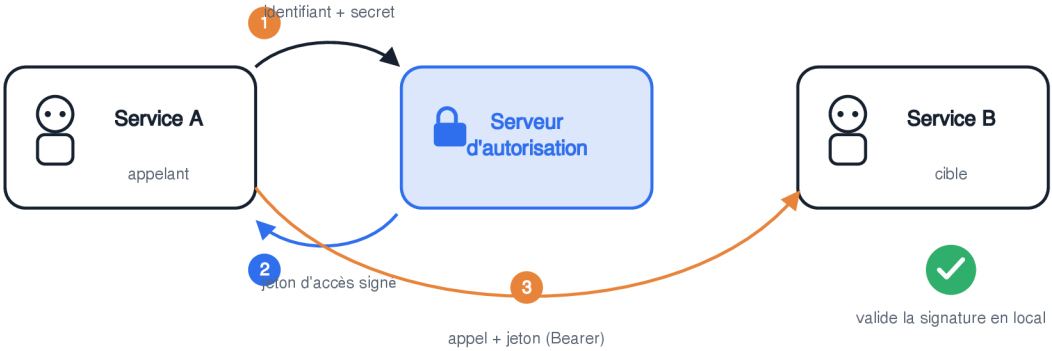
Une subtilité importante pour les développeurs : où le front stocke-t-il le jeton ? Retenez pour l'instant qu'aucune n'est neutre, et que ce choix doit être fait consciemment, pas par défaut.

05

Le JWT dans les échanges back / back

Quand ce sont les machines qui s'authentifient entre elles

Le second grand cas d'usage, souvent sous-estimé : la communication entre services, sans utilisateur humain dans la boucle. Une tâche planifiée qui appelle une API, un microservice qui en interroge un autre, une intégration partenaire.



Ici, il n'y a pas de mot de passe à saisir. Le standard de référence est le flux **OAuth 2.0 Client Credentials**. Le service appelant possède un identifiant et un secret ; il les présente à un serveur d'autorisation, qui lui délivre un jeton d'accès. Le service présente ensuite ce jeton au service cible, qui le valide exactement comme il validerait celui d'un utilisateur.

L'intérêt est le même que pour le front : **chaque service valide les jetons localement**, sans dépendre en temps réel du serveur d'autorisation. Cela rend le système résilient — si le serveur d'autorisation est momentanément indisponible, les jetons déjà émis continuent de fonctionner jusqu'à leur expiration — et performant, puisqu'on évite un appel réseau supplémentaire à chaque échange.

Pour les contextes les plus sensibles, on combine souvent le JWT avec une authentification mutuelle TLS (**mTLS**), où les deux services présentent un certificat. Le JWT porte alors les autorisations, le certificat garantit l'identité du canal.

06

Le cycle de vie d'un token

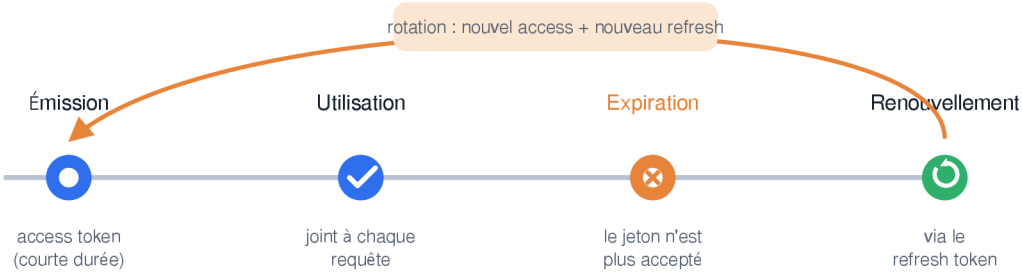
Court, renouvelable, révocable : l'art de gérer l'expiration

Un jeton ne vit pas éternellement, et c'est volontaire. La gestion de sa durée de vie est au cœur d'une implémentation saine. Le modèle recommandé repose sur **deux jetons distincts** :

L'access token est de courte durée (typiquement quelques minutes à une heure). C'est lui qui accompagne chaque requête. S'il est compromis, la fenêtre d'exploitation reste limitée.

Le refresh token est de plus longue durée et a un seul rôle : obtenir un nouvel access token quand celui-ci expire, sans forcer l'utilisateur à se reconnecter. Il est précieux, donc traité avec plus de précautions et stocké de façon plus restrictive.

Quand l'access token expire, le front présente discrètement le refresh token au back, reçoit un access token frais, et l'utilisateur ne s'aperçoit de rien.



La pratique la plus robuste est **la rotation des refresh tokens** : à chaque rafraîchissement, l'ancien refresh token est invalidé et un nouveau est émis. Si un refresh token déjà utilisé réapparaît, c'est le signe probable d'un vol — le système peut alors invalider toute la chaîne et forcer une reconnexion. On transforme ainsi une faiblesse potentielle en mécanisme de détection.

07

Les bonnes pratiques de sécurité

Les points non négociables qui séparent un JWT sûr d'une faille

C'est le cœur technique du document. Le JWT est sûr s'il est bien implémenté. La majorité des incidents proviennent d'erreurs évitables. Voici les points non négociables.

Algorithme imposé

Liste blanche côté serveur, « none » rejeté.

Durée de vie courte

Quelques minutes à une heure pour l'access token.

Valider tous les claims

exp, nbf, iss et surtout aud, à chaque requête.

Aucune donnée sensible

Le payload est lisible par tous. JWE si chiffrement.

HTTPS partout

Un jeton en clair peut être intercepté et rejoué.

Rotation des refresh tokens

Un refresh token par usage, avec détection de vol.

Stockage client réfléchi

Cookie HttpOnly vs localStorage : XSS vs CSRF.

Clés asymétriques

RS256 / ES256 / EdDSA dès que plusieurs valident.

Imposer l'algorithme côté serveur

Le serveur ne doit jamais faire confiance au champ **alg** du jeton reçu. Il doit définir une liste blanche d'algorithmes autorisés et rejeter tout le reste. C'est la parade à deux attaques classiques : **l'algorithme none** (un jeton non signé que des bibliothèques mal configurées acceptaient) et **la confusion d'algorithme** (où un attaquant fait passer une clé publique RSA pour un secret HMAC).

Choisir un algorithme adapté
Le **HMAC** (HS256) utilise un secret partagé : simple, mais tout service qui vérifie peut aussi forger des jetons. Les algorithmes asymétriques (RS256, ES256, EdDSA) séparent la clé privée qui signe de la clé publique qui vérifie : à privilégier dès que plusieurs services différents valident les jetons, ce qui est presque toujours le cas en microservices.

Valider systématiquement tous les claims
Vérifier la signature ne suffit pas. Il faut contrôler l'expiration (**exp**), l'éventuel « pas avant » (**nbf**), l'émetteur attendu (**iss**) et surtout l'audience (**aud**). Sans contrôle de l'audience, un jeton émis pour un service pourrait être rejoué contre un autre.

Garder les durées de vie courtes
Un access token de longue durée est une bombe à retardement. Quelques minutes à une heure constituent un bon compromis ; au-delà, le couple access/refresh token devient indispensable.

Ne jamais mettre de données sensibles dans le payload
Rappel essentiel : le contenu d'un JWT est *lisible* par tous. Pas de mot de passe, pas de numéro de carte, pas de donnée personnelle sensible. Si un chiffrement réel du contenu est nécessaire, c'est le standard JWE (*JSON Web Encryption*) qu'il faut utiliser, pas le **JWT signé classique**.

Toujours passer par HTTPS
Un jeton circulant en clair sur le réseau peut être intercepté et rejoué. Le chiffrement du transport n'est pas optionnel.

Choisir consciemment le stockage côté client
Deux options et leurs compromis :

- **localStorage :** simple d'accès en JavaScript, mais vulnérable aux attaques XSS — un script malveillant injecté dans la page peut lire le jeton.

- **Cookie HttpOnly, Secure, SameSite :** inaccessible au JavaScript (donc protégé du XSS), mais demande une protection contre les attaques CSRF.

Les limites : quand le JWT n'est pas la bonne réponse

Le prix de l'absence d'état, et comment l'assumer

Un livre blanc honnête doit aussi dire où la mécanique atteint ses limites. *Le JWT n'est pas une solution universelle.*

La révocation immédiate est complexe
C'est la contrepartie directe du modèle sans état. Un *access token* reste valide jusqu'à son expiration, même si l'utilisateur a été banni ou déconnecté entre-temps.

Les parades existent — durées de vie très courtes, liste de révocation (*blocklist*) des identifiants **jti**, versionnage des jetons par utilisateur — mais elles réintroduisent une part d'état, ce qui érode l'avantage initial.

Si votre besoin métier exige une révocation à la seconde, la session classique reste parfois plus simple.

La taille du jeton n'est pas neutre
Chaque requête transporte le jeton. Un payload chargé de nombreux **claims** alourdit toutes les requêtes et peut dépasser les limites de taille des en-têtes ou des cookies. Le payload doit rester minimal.
Ce n'est pas un mécanisme de chiffrement. Nous l'avons dit, mais cela mérite d'être répété, car c'est l'erreur conceptuelle la plus fréquente côté décideurs.

En résumé : le JWT excelle pour des autorisations à durée de vie courte, dans des architectures distribuées. Il est moins pertinent pour des sessions longues nécessitant un contrôle de révocation fin.

Checklist de mise en production

Ce qu'il faut cocher avant de déployer

- [ ] **Liste blanche d'algorithmes imposée côté serveur, None explicitement rejeté**
- [ ] **Algorithme asymétrique (RS256/ES256/EdDSA) si plusieurs services valident**
- [ ] **Validation de exp, nbf, iss et aud à chaque requête**
- [ ] **Access tokens de courte durée + refresh tokens avec rotation**
- [ ] **Détection de réutilisation des refresh tokens**
- [ ] **Aucune donnée sensible dans le payload**
- [ ] **HTTPS imposé sur tous les échanges**
- [ ] **Stratégie de stockage client choisie et documentée (XSS vs CSRF)**
- [ ] **Stratégie de révocation définie selon le besoin métier**
- [ ] **Clés de signature stockées dans un coffre, jamais dans le code**
- [ ] **Rotation des clés de signature planifiée**

Conclusion

Une clé de voûte solide, à condition de bien la poser

Le JWT s'est imposé parce qu'il répond à un besoin précis des architectures modernes : transmettre une identité vérifiable, sans état partagé, à travers des systèmes découplés. Côté décideurs, il se traduit par une meilleure scalabilité et un découplage propre entre services. Côté développeurs, il offre un standard outillé, documenté et interopérable.

Mais sa simplicité apparente est trompeuse : la sécurité d'une implémentation JWT tient entièrement dans la rigueur de sa validation et de sa gestion du cycle de vie. Bien conçu, c'est une clé de voûte solide. Mal conçu, c'est une porte ouverte.

L'objectif de ce document était de vous donner les deux perspectives pour faire ce choix en connaissance de cause.

Ce livre blanc a été rédigé par les équipes de ChallengeMyProject.

David Patiashvili

Fondateur, président & CTPO

E-mail :
david.patiashvili
@challengemyproject.com

Téléphone :
+ 33 (7) 69 40 60 43

Visio / RDV :
challengemyproject.com

David Patiashvili
Fondateur | Directeur technique

+33 (0)7 69 40 60 43
david.patiashvili@challengemyproject.com