Documentation complète sur JWT (JSON Web Token)

1. Introduction

JWT (JSON Web Token) est un standard ouvert (RFC 7519) qui définit une méthode compacte et sécurisée pour transmettre des informations entre parties sous forme d'objet JSON. Ces informations peuvent être vérifiées et approuvées car elles sont signées numériquement.

2. Structure d'un JWT

Un JWT est composé de **trois parties** séparées par des points (`.`) :

```
header.payload.signature
```

2.1 Header (En-tête)

Contient des métadonnées sur le token :

- **alg** : Algorithme de signature (ex: HS256, RS256)
- **typ** : Type de token (JWT)

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

2.2 Payload (Charge utile)

Contient les **claims** (revendications) - les données à transmettre :

Types de claims :

- **Registered claims** : Standards prédéfinis
  - `iss` (issuer) : Émetteur du token
  - `sub` (subject) : Sujet du token
  - `aud` (audience) : Destinataire
  - `exp` (expiration) : Date d'expiration
  - `nbf` (not before) : Date avant laquelle le token n'est pas valide
  - `iat` (issued at) : Date d'émission
  - `jti` (JWT ID) : Identifiant unique
- **Public claims** : Définis dans le registre IANA ou avec une URI
- **Private claims** : Personnalisés entre les parties

```json
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022,
  "exp": 1516325422,
  "role": "admin"
}
```

2.3 Signature

Garantit l'intégrité du token. Créée en combinant :

- Header encodé
- Payload encodé
- Secret (ou clé privée)
- Algorithme spécifié dans le header

```javascript
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

3. Processus de fonctionnement

3.1 Génération du JWT

1. L'utilisateur s'authentifie (login/password)
2. Le serveur vérifie les credentials
3. Si valides, le serveur génère un JWT
4. Le JWT est retourné au client

3.2 Utilisation du JWT

1. Le client stocke le JWT (localStorage, cookie, etc.)
2. Pour chaque requête protégée, le client envoie le JWT
3. Le serveur vérifie la signature du JWT
4. Si valide, le serveur traite la requête

4. Avantages et inconvénients

✅ Avantages

- **Compact** : Format léger, idéal pour les URLs et headers HTTP
- **Autonome** : Contient toutes les informations nécessaires
- **Sécurisé** : Signé numériquement
- **Interopérable** : Standard JSON, compatible multiplateforme
- **Stateless** : Pas besoin de stocker les sessions côté serveur
- **Scalable** : Facilite la montée en charge

❌ Inconvénients

- **Révocation difficile** : Impossible d'invalider un JWT avant expiration
- **Taille** : Plus volumineux qu'un simple ID de session
- **Sécurité** : Si compromis, valide jusqu'à expiration
- **Données sensibles** : Le payload est lisible (base64, pas chiffré)

5. Sécurité et bonnes pratiques

🔒 Bonnes pratiques de sécurité

5.1 Stockage côté client

- **Recommandé** : httpOnly cookies (protection XSS)
- **Éviter** : localStorage (vulnérable aux attaques XSS)

5.2 Durée de vie

- **Access tokens** : Courte durée (15-30 minutes)

- **Refresh tokens** : Plus longue durée, stockés sécurisément
- Implémenter un système de refresh automatique

5.3 Algorithmes de signature

- **Éviter** : Algorithme "none"
- **Recommandé** : HS256 (HMAC) ou RS256 (RSA)
- Toujours vérifier l'algorithme côté serveur

5.4 Validation

- Vérifier la signature
- Contrôler l'expiration (`exp`)
- Valider l'émetteur (`iss`) et l'audience (`aud`)
- Vérifier les claims personnalisés

6. Implémentation pratique

6.1 Exemple avec Node.js (jsonwebtoken)

javascript

```javascript
const jwt = require('jsonwebtoken');

// Génération d'un JWT
const generateToken = (user) => {
  const payload = {
    sub: user.id,
    email: user.email,
    role: user.role,
    iat: Math.floor(Date.now() / 1000),
    exp: Math.floor(Date.now() / 1000) + (60 * 60) // 1 heure
  };

  return jwt.sign(payload, process.env.JWT_SECRET);
};

// Vérification d'un JWT
const verifyToken = (token) => {
  try {
    return jwt.verify(token, process.env.JWT_SECRET);
  } catch (error) {
    throw new Error('Token invalide');
  }
};
```

6.2 Middleware d'authentification

javascript

```javascript
const authenticateToken = (req, res, next) => {
  const authHeader = req.headers['authorization'];
  const token = authHeader && authHeader.split(' ')[1]; // Bearer TOKEN

  if (!token) {
    return res.status(401).json({ error: 'Token requis' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(403).json({ error: 'Token invalide' });
  }
};
```

6.3 Utilisation côté client (JavaScript)

javascript

```javascript
// Envoi du JWT dans les requêtes
const makeAuthenticatedRequest = async (url, token) => {
  const response = await fetch(url, {
    headers: {
      'Authorization': `Bearer ${token}`,
      'Content-Type': 'application/json'
    }
  });

  return response.json();
};

// Décodage du payload (sans vérification)
const decodeToken = (token) => {
  const payload = token.split('.')[1];
  return JSON.parse(atob(payload));
};
```

7. JWT vs Sessions traditionnelles

Critère	JWT	Sessions
Stockage serveur	Non requis	Base de données/mémoire
Scalabilité	Excellente	Limitée
Sécurité	Dépend de l'implémentation	Plus facile à sécuriser
Révocation	Difficile	Facile
Taille	Plus volumineux	ID simple
Stateless	Oui	Non

8. Cas d'usage recommandés

✅ Quand utiliser JWT

- Applications distribuées/microservices
- API REST stateless
- Single Sign-On (SSO)
- Applications mobile
- Autorisation entre services

❌ Quand éviter JWT

- Applications avec besoin de révocation fréquente
- Données très sensibles
- Applications monolithiques simples
- Sessions de longue durée

9. Outils et librairies

Librairies populaires

- **Node.js** : jsonwebtoken, jose
- **Python** : PyJWT, python-jose
- **Java** : jjwt, java-jwt
- **PHP** : firebase/php-jwt
- **C#** : System.IdentityModel.Tokens.Jwt

Outils de debug

- jwt.io : Décodeur/encodeur en ligne
- jwt-cli : Outil en ligne de commande

10. Conclusion

JWT est un excellent choix pour l'authentification dans les architectures modernes, particulièrement les API REST et les applications distribuées. Cependant, une implémentation sécurisée nécessite une attention particulière aux bonnes pratiques de sécurité, notamment concernant le stockage, la durée de vie des tokens, et la validation côté serveur.