Api & Sécurité

Auteur : Cyril Vimard

Le Monde de la Cybersécurité aujourd’hui

Dans un monde de plus en plus interconnecté, les API (Interfaces de Programmation d'Applications) jouent un rôle clé en permettant aux différents systèmes et applications de communiquer entre eux.

Elles sont au cœur des services modernes, que ce soit pour les applications mobiles, les sites web, ou encore les objets connectés.

Cependant, les API peuvent également devenir des cibles privilégiées pour les attaquants cherchant à exploiter des vulnérabilités. C’est pourquoi comprendre la sécurité des API est essentiel pour protéger les données et les services dans l’environnement numérique actuel.

Le Monde de la Cybersécurité aujourd’hui

Les conséquences d'une faille de sécurité sur une API peuvent être dramatiques :

Exposition de données sensibles : Les API peuvent permettre l'accès à des informations sensibles comme des données personnelles, financières ou des informations d’identification.

Prise de contrôle de l'application : Si un attaquant parvient à exploiter une API mal sécurisée, il pourrait prendre le contrôle de l'application entière, menant à un vol ou une destruction de données.

Perturbation du service : Les attaques DDoS peuvent rendre une API inutilisable pendant un certain temps, entraînant une perte de disponibilité pour les utilisateurs.

Atteinte à la réputation : Une violation de données ou une faille de sécurité dans une API peut avoir des conséquences graves sur la réputation d'une organisation.

Les Risques de sécurité associés aux API

Vol de données sensibles

Vol de données sensibles

Les API, si elles ne sont pas protégées correctement, peuvent être utilisées pour accéder à des données sensibles (mots de passe, informations personnelles, données financières).

Exécution de commandes non autorisées

Les API mal sécurisées peuvent permettre aux attaquants d'exécuter des commandes non autorisées, modifiant ainsi les données ou les systèmes.

Attaques par injection

Par exemple, l'injection SQL ou d'autres types d'injections dans les requêtes API peuvent compromettre la base de données et exfiltrer des informations sensibles.

Vol de données sensibles

Dénis de service (DoS)

Les API peuvent être attaquées par des attaques par déni de service distribué (DDoS) qui saturent le serveur avec des requêtes malveillantes, empêchant son bon fonctionnement

Manque de contrôle d'accès

Si les API ne gèrent pas correctement l'accès, des utilisateurs non autorisés peuvent accéder à des données ou fonctionnalités sensibles.

Bonnes pratiques : **Authentification et Autorisation**

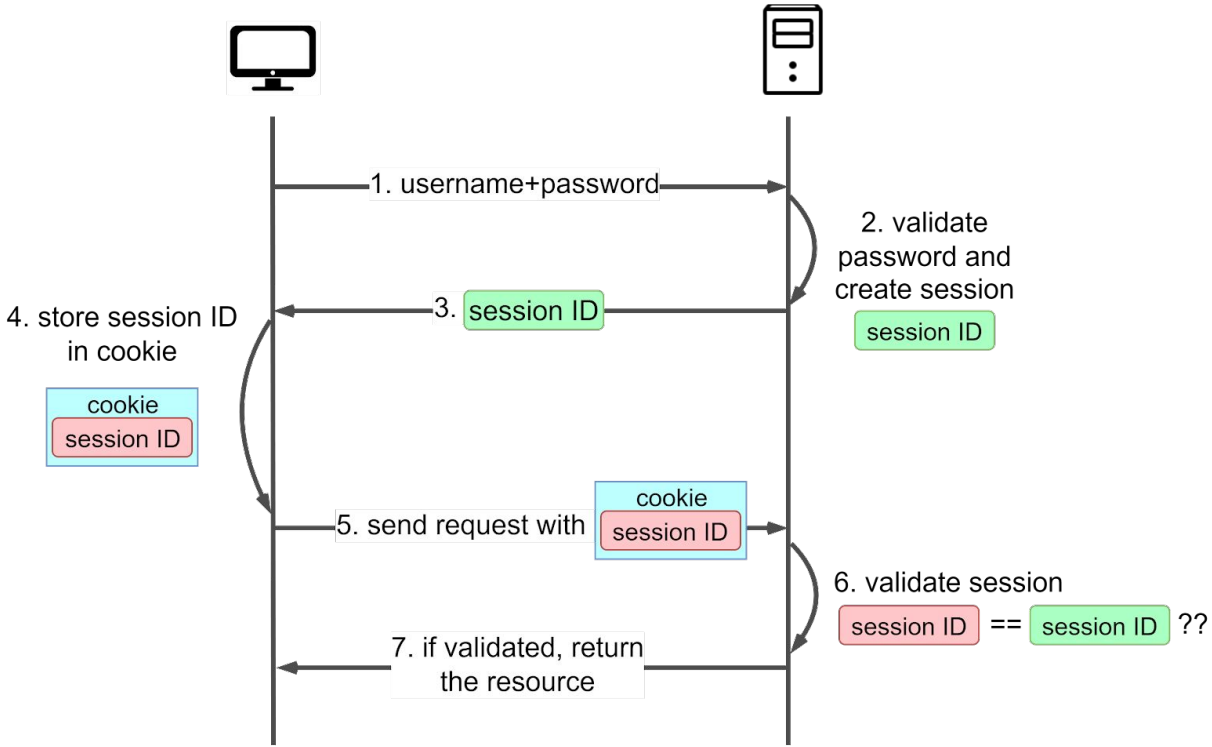
Authentification et Autorisation

JWT (JSON Web Tokens) :
JWT est un standard ouvert qui permet de transmettre des informations entre deux parties de manière sécurisée. Ces informations sont souvent utilisées pour l'authentification et l'autorisation des utilisateurs.

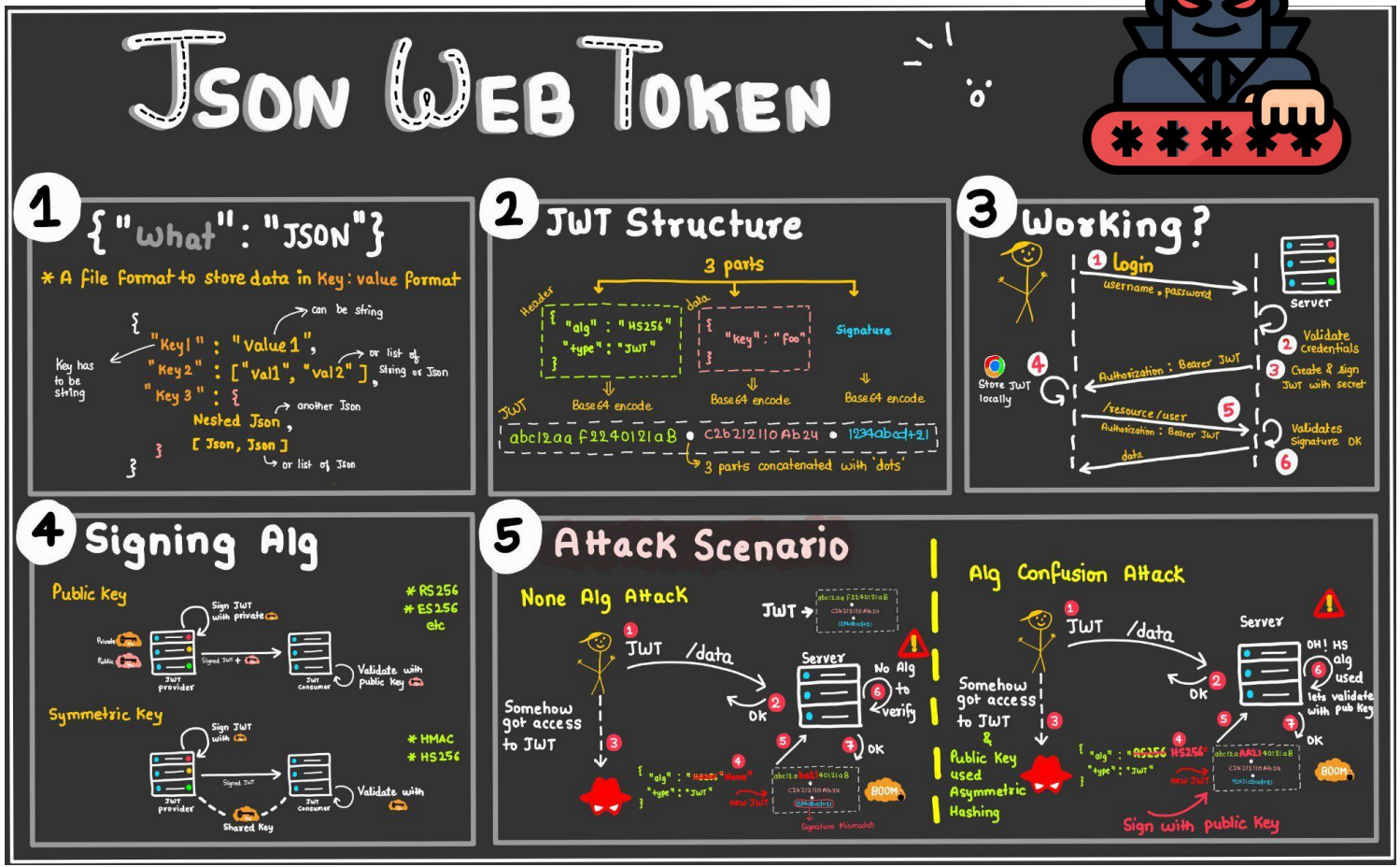
API Keys :
Une clé d'API permet d'identifier un utilisateur ou un service qui effectue une requête à l'API. Elle est souvent utilisée pour contrôler l'accès à une API.

MFA (Authentification Multi-facteurs) :
Ajoutez une couche supplémentaire de sécurité en exigeant que les utilisateurs passent par une validation de deux facteurs lors de l'authentification.

Exemple d'une authentification par cookie



Exemple d'une authentification avec JWT



Bonnes pratiques : **Contrôle des accès**

Contrôle des accès

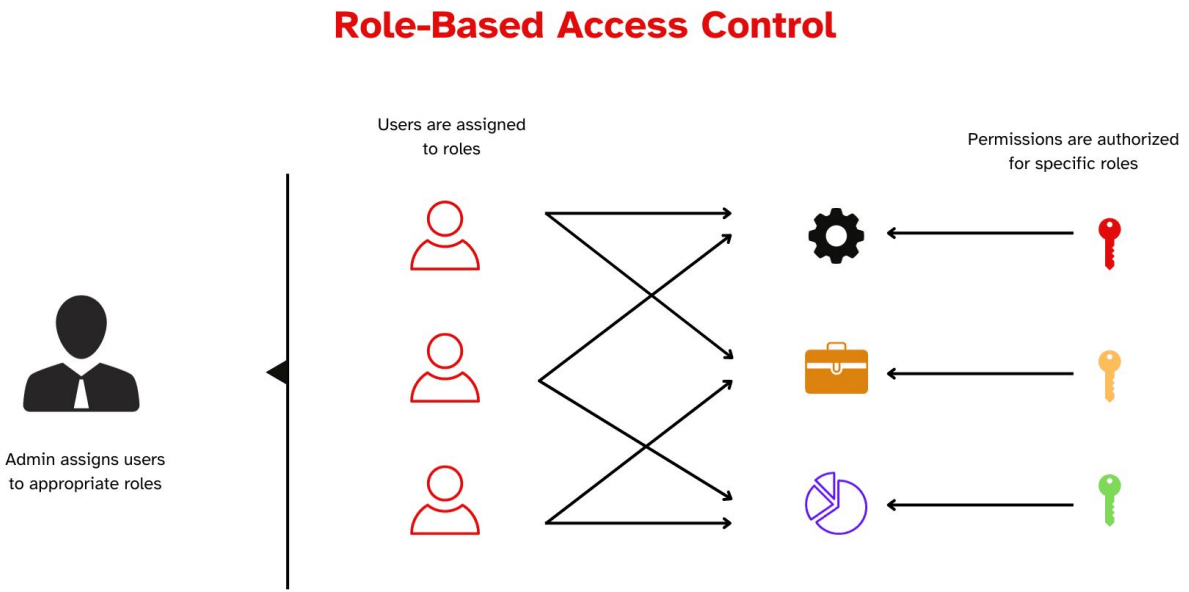
Contrôle d'accès basé sur les rôles (RBAC) :

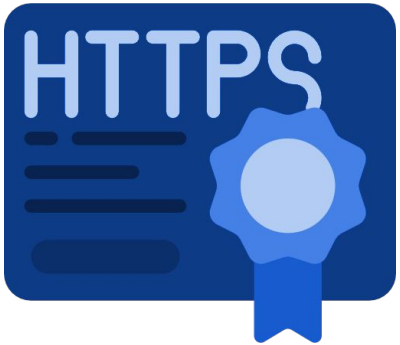
Limiter les actions qu'un utilisateur ou une application peut effectuer selon son rôle.

Listes de contrôle d'accès (ACL) :

Permet de spécifier précisément quels utilisateurs ou systèmes peuvent accéder à certaines API ou à certaines parties d'une API.

Contrôle des accès





Bonnes pratiques : **Chiffrement des communications**

Chiffrement des communications

HTTPS :

Toujours utiliser HTTPS pour garantir que les communications entre le client et le serveur API sont sécurisées et que les données ne peuvent pas être interceptées ou manipulées par un attaquant.

Chiffrement des données sensibles :

Assurez-vous que les informations sensibles sont chiffrées non seulement lors de la transmission, mais aussi au repos dans les bases de données.

Comment fonctionne HTTPS ?

HTTPS repose sur le chiffrement et utilise des technologies comme SSL/TLS pour assurer la sécurité des communications.

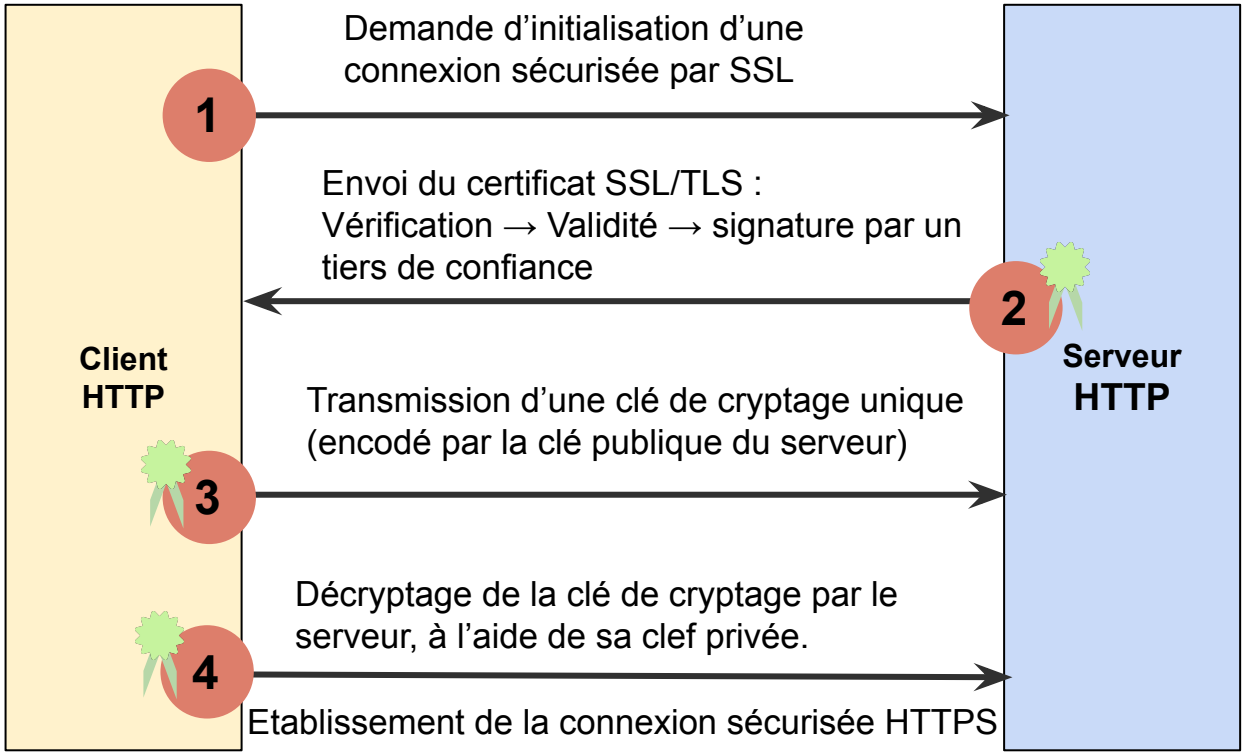
SSL/TLS : Le mécanisme sous-jacent SSL (Secure Sockets Layer) est un protocole de sécurité utilisé pour chiffrer les communications sur un réseau. Cependant, il a été remplacé par une version plus sécurisée appelée TLS (Transport Layer Security).

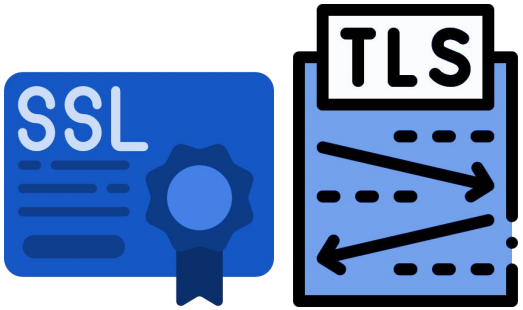
TLS est la version actuelle utilisée dans HTTPS pour sécuriser les communications.

Il offre des fonctionnalités comme :

- **Chiffrement** : Chiffre les données échangées entre le client et le serveur.
- **Authentification** : Garantit que le serveur est bien celui qu'il prétend être, grâce à l'utilisation de certificats numériques.
- **Intégrité** : S'assure que les données transmises ne sont pas modifiées pendant leur transfert.

HTTPS


Demande d'initialisation d'une connexion sécurisée par SSL
1
Envoi du certificat SSL/TLS : Vérification → Validité → signature par un tiers de confiance
2
Client HTTP
Serveur HTTP
Transmission d'une clé de cryptage unique (encodé par la clé publique du serveur)
3
Décryptage de la clé de cryptage par le serveur, à l'aide de sa clef privée.
4
Etablissement de la connexion sécurisée HTTPS



Certificat SSL/TLS

Comment générer une clé SSL pour HTTPS ?

Étape 1 : Installer les outils nécessaires

sudo apt install openssl nginx

Étape 2 : Générer une clé privée

openssl genrsa -out /etc/ssl/private/nginx.key 2048

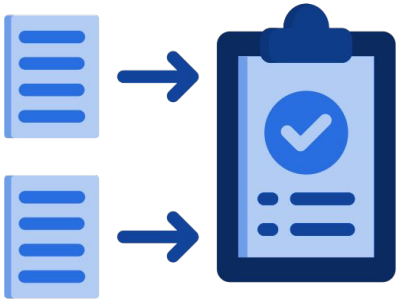
Étape 3 : Générer une demande de signature de certificat (CSR)

openssl req -new -key /etc/ssl/private/nginx.key -out /etc/ssl/private/nginx.csr

Étape 4 : Générer un certificat auto-signé

openssl x509 -req -days 365 -in /etc/ssl/private/nginx.csr -signkey /etc/ssl/private/nginx.key -out /etc/ssl/certs/nginx.crt

Étape 5 : Configurer votre api pour utiliser le certificat



Bonnes pratiques : **Validation des entrées et prévention des attaques par injection**

Validation des entrées et prévention des attaques par injection

Validation des entrées :

Ne jamais faire confiance aux données provenant de l'extérieur. Validez soigneusement toutes les entrées, y compris les paramètres d'URL, les en-têtes HTTP et les corps de requêtes.

Protection contre l'injection SQL :

Utiliser des requêtes préparées pour empêcher l'injection SQL et d'autres types d'injections.

Sanitisation des entrées :

Assurez-vous que les entrées utilisateur sont nettoyées et validées pour éviter les attaques XSS (Cross-Site Scripting) ou CSRF (Cross-Site Request Forgery).

Bonnes pratiques : **Limitation du taux de requêtes (Rate Limiting)**

Limitation du taux de requêtes (Rate Limiting)

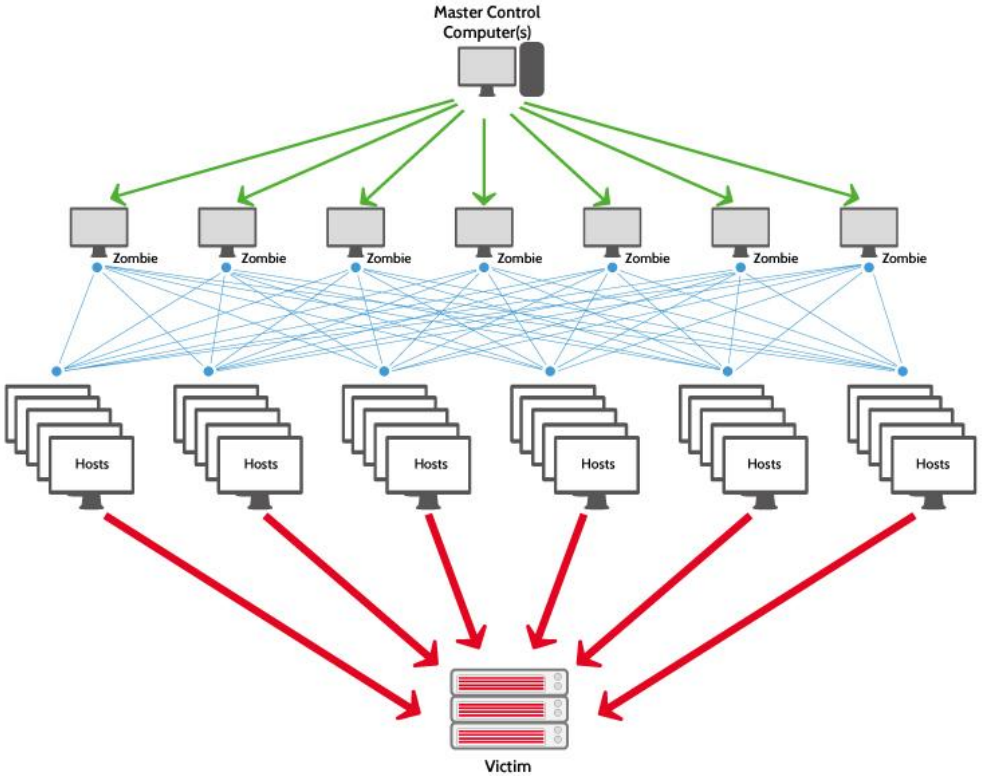
Limiter le nombre de requêtes :

Implémenter des mécanismes de rate limiting pour empêcher les attaques par déni de service (DoS) et limiter l'impact d'un attaquant qui essaierait d'envoyer des requêtes excessives.

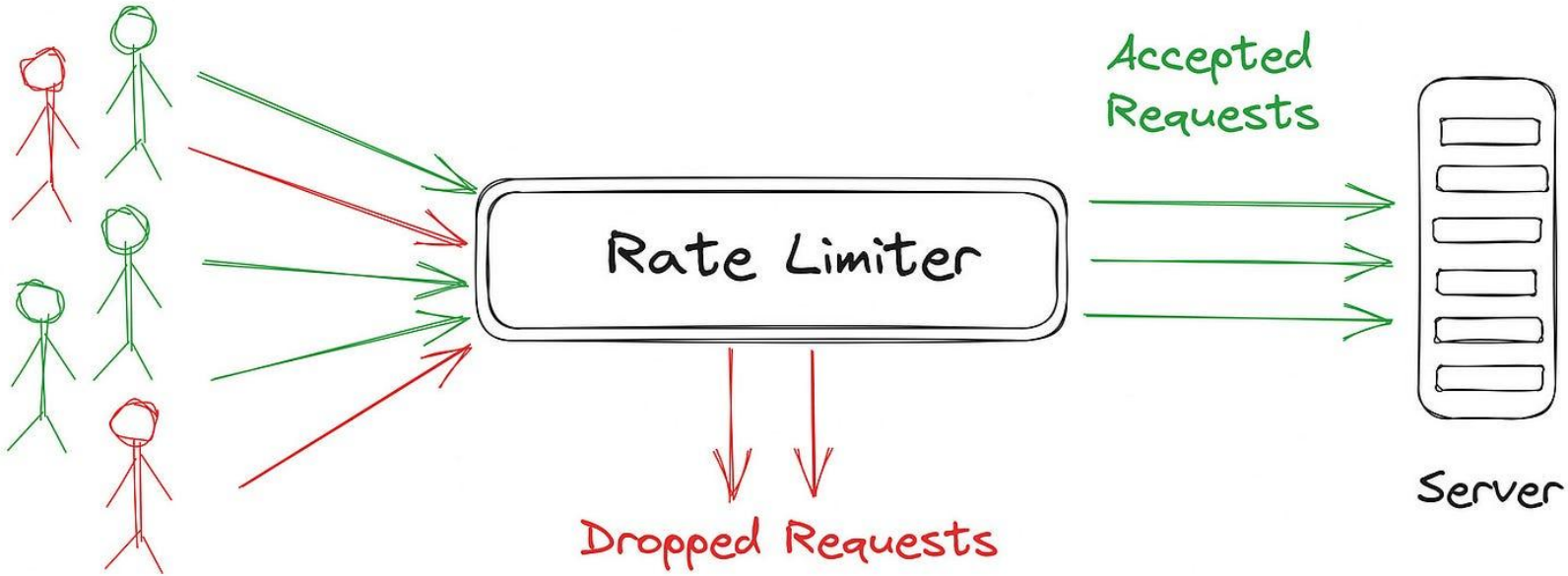
CAPTCHA :

Intégrer un mécanisme CAPTCHA pour vérifier que les requêtes proviennent de vrais utilisateurs et non de robots.

Limitation du taux de requêtes (Rate Limiting)



Limitation du taux de requêtes (Rate Limiting)



Bonnes pratiques : **Surveillance et audit**

Surveillance et audit

Journaux d'accès (Logs) :

Conservez des journaux détaillés des accès aux API, y compris l'adresse IP de l'utilisateur, l'heure, les types de requêtes effectuées, et les résultats.

Audits réguliers :

Effectuez des audits de sécurité réguliers pour vérifier que les API respectent les normes de sécurité et qu’aucune vulnérabilité n’a été introduite.

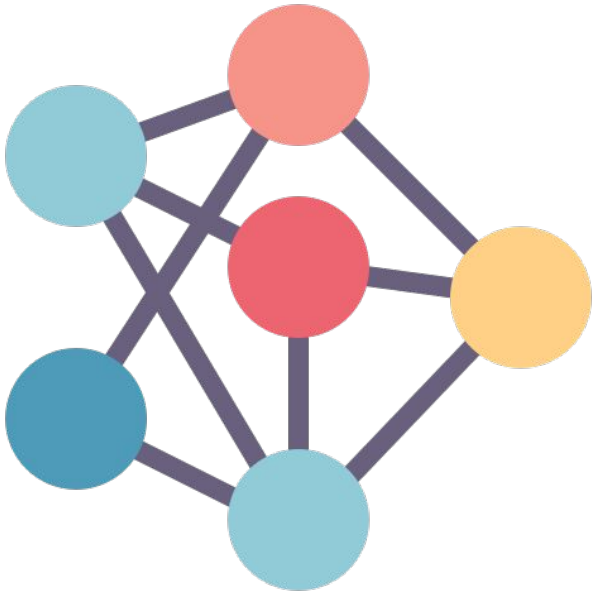
Le **CORS**

Le **CORS**

Le CORS **(Cross-Origin Resource Sharing)** est un mécanisme utilisé par les navigateurs web pour permettre ou restreindre les requêtes HTTP provenant de domaines différents **(cross-origin)**.

Il s'agit d'un élément clé de la sécurité des applications web modernes, car il empêche les scripts malveillants sur un site d'accéder à des ressources sur un autre domaine sans autorisation.

En règle générale, **les navigateurs web bloquent les requêtes HTTP effectuées d'un domaine (ou origine) vers un autre domaine si elles ne sont pas explicitement autorisées**. Ce mécanisme fait partie de la politique de sécurité du même domaine (Same-Origin Policy) qui est conçue pour protéger les utilisateurs des attaques, comme le vol de données sensibles.



Le **CORS**

Origine (ou Domaine)

L'origine est définie par une combinaison de :
- Le schéma (protocole utilisé : **http**, **https**, **ftp**, etc.)
- Le nom de domaine (ex: **example.com**)
- Le port (ex: :**80**, :**443**)

Deux ressources ont la même origine si ces trois composants sont identiques.

Same-Origin Policy (SOP)

La Same-Origin Policy **empêche qu'un script sur une page web issue d'une certaine origine puisse interagir librement avec des ressources** (API, données, etc.) d'une autre origine.

Par exemple : Une page web chargée depuis https://www.example.com ne pourra pas faire des requêtes AJAX vers https://api.anotherdomain.com sans autorisation explicite.

Le besoin de CORS

Cependant, ***de nombreuses applications web modernes utilisent des architectures où le frontend (client) et le backend (serveur) sont hébergés sur des domaines différents***.

Par exemple, une application SPA (Single Page Application) peut être hébergée sur https://frontend.com et elle doit communiquer avec une API sur https://api.backend.com.

Dans ce cas, le navigateur bloque par défaut ces requêtes cross-origin. CORS est utilisé pour lever cette restriction sous certaines conditions.

Comment fonctionne le **CORS** ?

Les étapes d'une requête **CORS**

Il permet au serveur de spécifier **quelles origines sont autorisées à accéder à ses ressources**. Lorsque le navigateur détecte une requête cross-origin, il envoie des en-têtes CORS au serveur, qui, à son tour, répond avec des en-têtes spécifiant si la requête est autorisée ou non.

Les étapes d'une requête CORS

1/ Requête Simple (Simple Request) :

- Une requête simple inclut des requêtes de type **GET** ou **POST** sans en-tête particulier (autres que ceux courants comme **Content-Type**).
- Exemple : **un script JavaScript sur https://frontend.com effectue une requête GET vers https://api.backend.com/data.**
- Le serveur backend doit répondre avec l'en-tête Access-Control-Allow-Origin pour permettre la requête. Par exemple : **Access-Control-Allow-Origin: https://frontend.com**
- Si cet en-tête n'est pas présent ou si la valeur ne correspond pas à l'origine de la requête, le navigateur bloque la réponse.

2/ Requête Prévol (Preflight Request) :

- Pour certaines requêtes plus complexes (méthodes HTTP autres que GET/POST, ou requêtes avec des en-têtes personnalisés), le navigateur effectue d'abord une requête OPTIONS pour vérifier si l'origine est autorisée avant de procéder à la requête réelle. C'est ce qu'on appelle une "pré-requête" ou requête prévol.
- Exemple de requête OPTIONS envoyée par le navigateur : **OPTIONS /data HTTP/1.1 Origin: https://frontend.com Access-Control-Request-Method: PUT**
- Le serveur doit répondre avec les bons en-têtes, comme : **Access-Control-Allow-Origin: https://frontend.com Access-Control-Allow-Methods: GET, POST, PUT, DELETE** Access-Control-Allow-Headers: **Content-Type Access-Control-Max-Age: 86400 (cache la réponse prévol pendant 1 jour)**
- Si la réponse est favorable, la requête réelle est envoyée après la validation du prévol.

Principaux en-têtes **CORS**

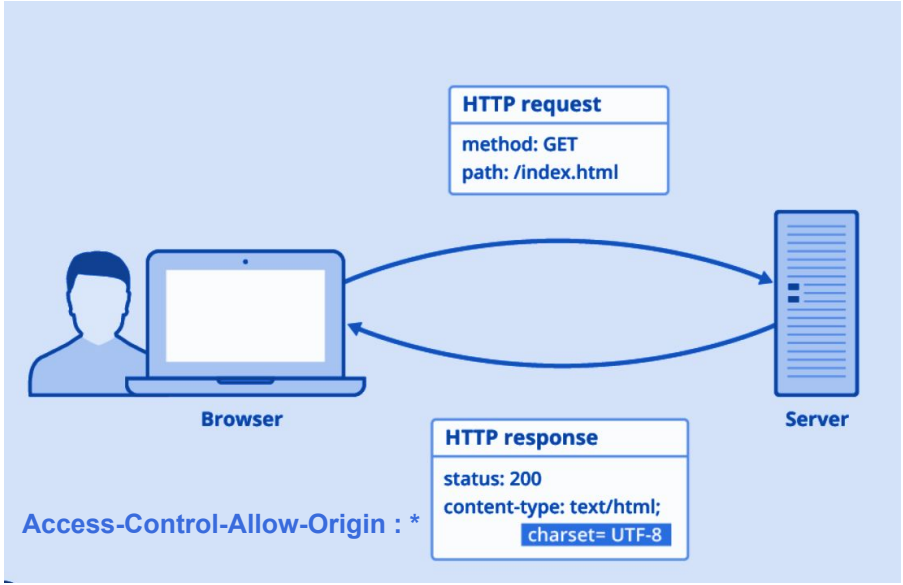
Access-Control-Allow-Origin : Spécifie quelles origines peuvent accéder aux ressources. Peut être un domaine spécifique (https://frontend.com) ou un joker (*) qui permet à toutes les origines d'accéder.

Access-Control-Allow-Methods : Liste des méthodes HTTP autorisées (ex: GET, POST, PUT).

Access-Control-Allow-Headers : Spécifie quels en-têtes peuvent être utilisés dans la requête (ex: Authorization, Content-Type).

Access-Control-Allow-Credentials : Autorise l'envoi de cookies ou d'informations d'authentification dans une requête (valeur true).

Access-Control-Max-Age : Indique combien de temps (en secondes) le résultat d'une pré-requête peut être mis en cache par le navigateur.



CORS et Sécurité

CORS et **Sécurité**

Le but de CORS est de renforcer la sécurité des échanges inter-domaine, mais une mauvaise configuration peut être source de vulnérabilités :

Dangers d'une mauvaise configuration

Access-Control-Allow-Origin: * : Permet à toutes les origines d'accéder aux ressources, ce qui peut être dangereux si utilisé sans précaution.

Access-Control-Allow-Credentials: **true** avec un **Access-Control-Allow-Origin**: * : Cela constitue une faille de sécurité potentielle, car cela permet l'envoi de cookies ou de tokens d'authentification à toutes les origines, même non autorisées.

Bonnes pratiques

- Restreindre les origines autorisées spécifiquement.
- Ne jamais utiliser **Access-Control-Allow-Origin: * avec Access-Control-Allow-Credentials: true**.
- Tester et valider les configurations de sécurité CORS avec des outils comme **curl**, **Postman** ou via les consoles de navigateurs pour s'assurer que seules les origines légitimes ont accès.

[Api & sécurité]
Workshop

Api & sécurité

Le but de ce TP est de sécuriser une API réalisée avec Node.js et Express en appliquant des techniques de sécurité telles que l'authentification et l'autorisation via JWT, la validation des entrées, la sécurisation des communications avec HTTPS, la limitation des requêtes, et la gestion des erreurs. Le tp est disponible ici

Prérequis

- **Avoir une API de base déjà réalisée avec Node.js et Express.**