

Chapitre 8 - Sécurité et bonnes pratiques

- [L'authentification](#)
- [HTTPS et CORS](#)
- [Le rate limiting](#)
- [L'OWASP API Security Top 10](#)

L'authentification

Authentification ou autorisation ?

- **Authentification** : *qui es-tu ?* L'échec donne un `401 Unauthorized` (mal nommé : il devrait s'appeler « Unauthenticated »).
- **Autorisation** : *as-tu le droit ?* L'échec donne un `403 Forbidden`.

Au bar : montrer sa carte de membre, c'est l'authentification. Le videur qui vérifie que la carte donne accès au carré VIP, c'est l'autorisation.

HTTP étant sans état, le client doit prouver son identité à **chaque requête**, en général dans le header `Authorization`.

La clé d'API

Une longue chaîne aléatoire, propre à chaque client, envoyée dans un header.

```
curl http://api-ritif.portlatch.com/v1/me -H "X-API-Key: ak_7f3c9e1b2d4a"
```

- ☐ Très simple, idéale pour les échanges entre serveurs et les scripts.
- ☐ Elle ne périmé pas toute seule : si elle fuit, elle reste valable jusqu'à sa révocation.
- ☐ Elle identifie une application plus qu'un utilisateur.

Une clé d'API est un mot de passe : jamais dans le code, jamais dans git, jamais dans une URL (`?api_key=...` finit dans les logs et l'historique du navigateur). Côté serveur, on la stocke **hachée**, comme un mot de passe.

Basic Auth

L'identifiant et le mot de passe, encodés en Base64, dans le header `Authorization`.

```
curl -u neo:redpill https://api.example.com/v1/me
# envoie : Authorization: Basic bmVvOnJlZHBpbGw=
```

Base64 est un **encodage**, pas un chiffrement : `echo bmVv0nJlZHBpbGw= | base64 -d` affiche `neo:redpill`. Sans HTTPS, c'est un mot de passe en clair.

Sessions ou tokens ?

	Session (cookie)	Token (JWT...)
Principe	Le serveur garde la session en mémoire ou en base, le client garde un identifiant dans un cookie	Le client garde un jeton qui contient lui-même les informations, signé par le serveur
Envoi	Automatique par le navigateur (Cookie)	Manuel (Authorization: Bearer ...)
État côté serveur	Oui	Non : le serveur vérifie juste la signature
Révocation	Facile : on supprime la session	Difficile : le token reste valable jusqu'à son expiration
Adapté à...	Un site web classique	Une API, une application mobile, des microservices

JWT

Un **JSON Web Token** est composé de trois parties en Base64url, séparées par des points :

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMiIsIm5pY2tuYW1lIjoibmVvIiwicm9sZSI6ImNlc3RvbWVyIiwiaWF0eSkiOjEwMDAwfQ..7xq...  
└──────── header ─────────┬─────────── payload  
                             └─────────── signature
```

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

// payload (les « claims »)
{"sub": "12", "nickname": "neo", "role": "customer", "exp": 1794644000}
```

- Le **payload** est lisible par n'importe qui : **aucun secret dedans**.
- La **signature** garantit qu'il n'a pas été modifié : changer `"role": "customer"` en `"bartender"` invalide la signature.
- `exp` fixe la date d'expiration (en secondes depuis 1970). Un token expiré donne un `401`.

Pour lire un payload :

```
import base64, json
json.loads(base64.urlsafe_b64decode(token.split(".")[1] + "=="))
```

Le site jwt.io décode les JWT visuellement. Ne collez jamais un vrai token de production dans un site tiers.

Avec API-ritif

```
# Échanger sa clé contre un JWT valable 15 minutes
curl -s -X POST http://api-ritif.portlatch.com/v1/auth/token -H "X-API-Key: $KEY"
# {"access_token": "eyJ...", "token_type": "Bearer", "expires_in": 900}

# L'utiliser
curl -i http://api-ritif.portlatch.com/v1/me -H "Authorization: Bearer $TOKEN"
```

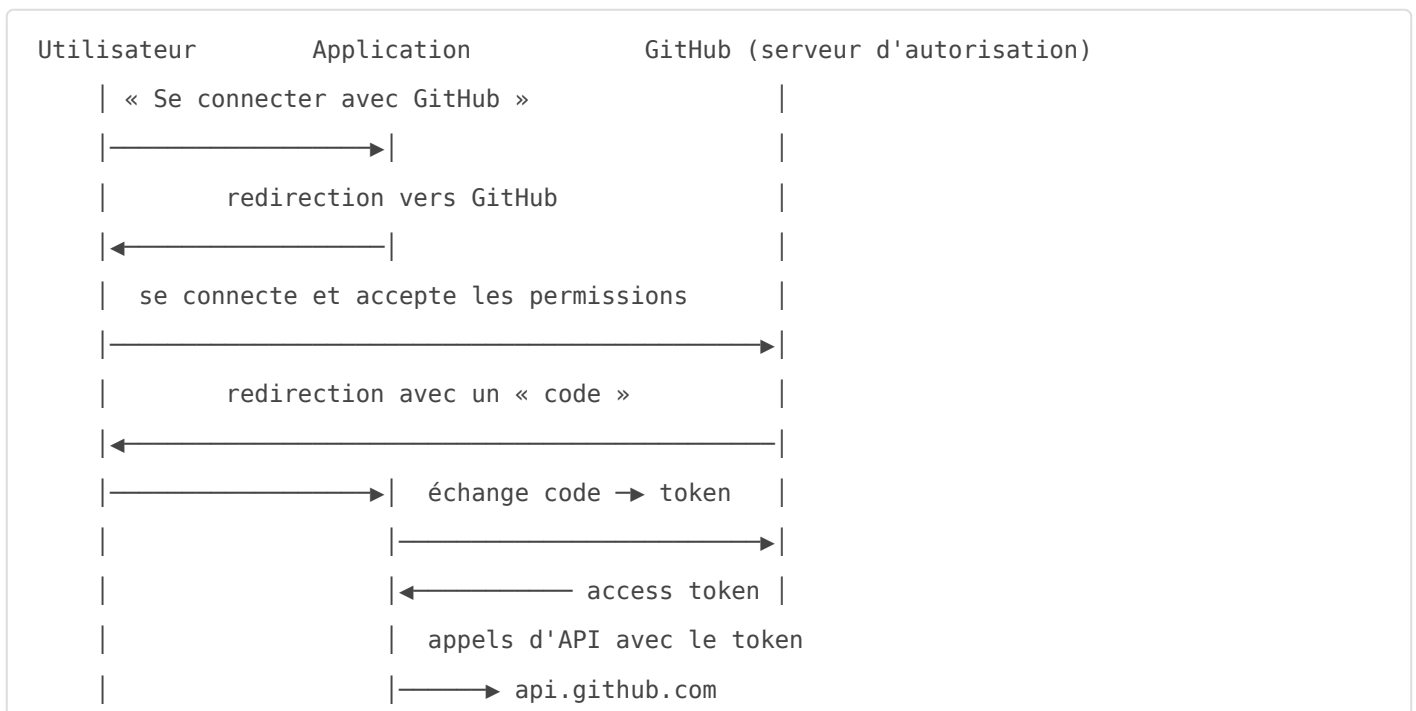
Revenez dans 15 minutes : même requête, `401`.

Les pièges

- `alg: none` : certaines vieilles bibliothèques acceptaient un token « non signé ». Le serveur doit imposer l'algorithme attendu.
- **Durée de vie** : courte (quelques minutes), avec un *refresh token* pour en obtenir un nouveau sans se reconnecter.
- **Stockage dans le navigateur** : `localStorage` est lisible par n'importe quel script de la page (faille XSS). Un cookie `HttpOnly; Secure; SameSite` est plus sûr.

OAuth2 : « Se connecter avec GitHub »

OAuth2 permet à une application d'agir **au nom d'un utilisateur** sur un autre service, sans jamais connaître son mot de passe.



- **Authorization Code + PKCE** : le flux ci-dessus, pour les applications web et mobiles.
- **Client Credentials** : pour un serveur qui appelle une API en son propre nom, sans utilisateur.
- **OpenID Connect** : une surcouche d'OAuth2 qui ajoute l'identité de l'utilisateur (un *ID token*, au format JWT).

On n'implémente jamais OAuth2 soi-même : on utilise un fournisseur (Keycloak, Auth0, Entra ID...) et une bibliothèque éprouvée.

HTTPS et CORS

HTTPS : non négociable

En HTTP, tout circule en clair : l'URL, les headers, le body. Sur le Wi-Fi de l'école, n'importe qui avec Wireshark peut lire votre clé d'API, votre token ou votre mot de passe.

HTTPS, c'est HTTP dans un tunnel **TLS**, qui apporte :

- la **confidentialité** : personne ne peut lire les échanges ;
- l'**intégrité** : personne ne peut les modifier en route ;
- l'**authentification du serveur** : le certificat prouve que vous parlez bien à `api-ritif.example`.

Avec Let's Encrypt, un certificat est gratuit et se renouvelle automatiquement. Il n'y a plus aucune excuse.

Pour une API, ne redirigez pas HTTP vers HTTPS : **refusez** les requêtes HTTP. Quand la redirection arrive, la clé d'API est déjà passée en clair, et le client ne s'en rendra jamais compte.

CORS

La politique de même origine

Par sécurité, un navigateur interdit au JavaScript d'une page de lire les réponses d'une autre **origine**. L'origine, c'est le trio **protocole + domaine + port** :

Page	Appelle	Même origine ?
<code>https://bar.example</code>	<code>https://bar.example/v1/cocktails</code>	☐
<code>https://bar.example</code>	<code>https://api.bar.example/v1/cocktails</code>	☐ domaine différent
<code>http://localhost:8080</code>	<code>http://localhost:3000/v1/cocktails</code>	☐ port différent
<code>http://bar.example</code>	<code>https://bar.example/v1/cocktails</code>	☐ protocole différent

Sans cette règle, n'importe quel site que vous visitez pourrait appeler, avec vos cookies, l'API de votre banque.

Le mécanisme CORS

CORS (*Cross-Origin Resource Sharing*) permet à une API d'**autoriser** certaines origines. Le navigateur envoie l'origine de la page dans le header `Origin`, et l'API répond si elle l'accepte :

```
GET /v1/cocktails HTTP/1.1
Origin: http://localhost:5173

HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://localhost:5173
```

Pour les requêtes « non simples » (un `DELETE`, du JSON, un header `X-API-Key`...), le navigateur envoie d'abord une requête **preflight** en `OPTIONS` pour demander la permission :

```
OPTIONS /v1/orders HTTP/1.1
Origin: http://localhost:5173
Access-Control-Request-Method: POST
Access-Control-Request-Headers: content-type, x-api-key

HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://localhost:5173
Access-Control-Allow-Methods: GET, POST, DELETE
Access-Control-Allow-Headers: content-type, x-api-key
```

CORS est une protection **du navigateur**. curl, Bruno ou un script Python ne l'appliquent pas : c'est pour ça que tout marchait jusqu'ici. Et c'est pour ça que CORS ne protège **pas** votre API contre un attaquant : seule l'authentification le fait.

Manip : provoquer une erreur CORS

API-ritif n'autorise qu'une seule origine. Saurez-vous trouver laquelle ?

1. Créez un fichier `index.html` :

```

<!doctype html>
<meta charset="utf-8">
<title>Test CORS</title>
<button id="go">Charger la carte</button>
<pre id="out"></pre>
<script>
  document.getElementById("go").onclick = async () => {
    const out = document.getElementById("out");
    try {
      const res = await fetch("http://api-ritif.portlatch.com/v1/cocktails");
      out.textContent = JSON.stringify(await res.json(), null, 2);
    } catch (e) {
      out.textContent = "Erreur : " + e.message;
    }
  };
</script>

```

2. Servez-le sur le port 8080 : `python3 -m http.server 8080`, ouvrez <http://localhost:8080> et cliquez. Lisez l'erreur dans la console des DevTools.
3. La même requête avec curl fonctionne. Comparez les headers de réponse avec et sans `Origin` :

```

curl -i http://api-ritif.portlatch.com/v1/cocktails -H "Origin:
http://localhost:8080"

```

4. Indice : les développeurs front qui utilisent Vite connaissent bien le port `5173`. Testez cette origine avec curl, puis servez la page sur ce port.
5. Modifiez le script pour passer une commande (`POST /v1/orders` avec du JSON et votre `X-API-Key`). Dans l'onglet Réseau, repérez la requête `OPTIONS` envoyée avant le `POST`.

Configurer CORS sur son API

```

from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173"],
    allow_methods=["*"],
    allow_headers=["*"],

```

)

`Access-Control-Allow-Origin: *` autorise tout le monde. C'est acceptable pour une API publique en lecture seule, jamais pour une API qui utilise des cookies.

Le rate limiting

Une API sans limite finit toujours par recevoir trop de requêtes : un script en boucle infinie, un client trop enthousiaste, un concurrent qui aspire vos données, ou une attaque par déni de service. Le **rate limiting** limite le nombre de requêtes par client et par période.

Exemple de réponse d'un barman débordé : « reviens dans 10 s ».

Pourquoi limiter ?

- **Disponibilité** : un seul client ne doit pas pouvoir monopoliser le serveur.
- **Coûts** : chaque requête coûte (calcul, base de données, appels à des services payants comme un LLM).
- **Sécurité** : ralentir les attaques par force brute sur une route de connexion.
- **Modèle économique** : offre gratuite à 60 requêtes par heure, offre payante à 5 000 (c'est le cas de l'API GitHub).

Les algorithmes

Algorithme	Principe
Fenêtre fixe	100 requêtes par minute, compteur remis à zéro à chaque minute. Simple, mais permet 200 requêtes en 2 secondes à cheval sur deux minutes.
Fenêtre glissante	100 requêtes sur les 60 dernières secondes, à tout instant. Plus juste.
Seau à jetons (<i>token bucket</i>)	Le client a un seau de jetons qui se remplit à vitesse constante. Chaque requête consomme un jeton. Autorise de petites rafales.

Ce que l'API doit répondre

```
HTTP/1.1 429 Too Many Requests
Retry-After: 10
Content-Type: application/problem+json
```

```
{"title": "Too Many Requests", "status": 429, "detail": "Le barman est débordé, reviens dans 10 s"}
```

Beaucoup d'API informent aussi le client **avant** qu'il atteigne la limite :

```
curl -i https://api.github.com/users/torvalds
```

```
x-ratelimit-limit: 60
x-ratelimit-remaining: 57
x-ratelimit-reset: 1794644000
```

Les headers `X-RateLimit-*` sont une convention répandue mais non standard. L'IETF travaille à un standard (`RateLimit` et `RateLimit-Policy`), déjà supporté par certaines bibliothèques.

Le mettre en place

```
# pip install slowapi
from fastapi import Request
from slowapi import Limiter
from slowapi.errors import RateLimitExceeded
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)
app.state.limiter = limiter

@app.exception_handler(RateLimitExceeded)
async def on_rate_limit(request: Request, exc: RateLimitExceeded):
    response = problem(429, "Too Many Requests", "Le barman est débordé, reviens dans une minute.")
    response.headers["Retry-After"] = "60"
    return response

@app.get("/v1/cocktails")
@limiter.limit("60/minute")
def list_cocktails(request: Request):    # le paramètre request est obligatoire
    ...
```

Limiter par adresse IP est un début, mais tous les étudiants d'une même salle partagent souvent la même IP publique. Une API authentifiée limite plutôt **par clé** ou par utilisateur.

En production, le rate limiting est souvent assuré en amont de l'application : par le reverse proxy (Nginx, Traefik), une passerelle d'API ou un CDN.

L'OWASP API Security Top 10

L'**OWASP** (*Open Worldwide Application Security Project*) est une fondation qui publie des références en sécurité applicative. Son **API Security Top 10** recense les risques les plus fréquents dans les API. La dernière édition date de 2023.

La plupart de ces failles ne demandent aucune compétence technique avancée pour être exploitées : juste curl et un peu de curiosité. Comme pendant la chasse au trésor.

#	Risque	Au bar, ça donne...	Se protéger
API1	Broken Object Level Authorization (BOLA, aussi appelé IDOR)	Changer l'id dans <code>/orders/42</code> et lire la commande d'un autre client	Vérifier à chaque requête que l'objet appartient à l'utilisateur
API2	Broken Authentication	Une route de connexion sans limite d'essais, des tokens qui n'expirent jamais	Bibliothèques éprouvées, expiration, rate limiting sur l'authentification
API3	Broken Object Property Level Authorization (BOPLA)	Envoyer <code>{"role": "bartender"}</code> dans un <code>PATCH /me</code> et devenir barman	N'accepter et ne renvoyer que les champs prévus (schémas stricts)
API4	Unrestricted Resource Consumption	<code>?per_page=1000000</code> , ou 10 000 commandes par seconde	Limites de pagination, de taille de body, rate limiting
API5	Broken Function Level Authorization	Un client qui appelle <code>/admin/scoreboard</code> et obtient une réponse	Contrôler le rôle sur chaque route sensible, refuser par défaut
API6	Unrestricted Access to Sensitive Business Flows	Un bot qui commande tous les 404 Spritz dès leur retour en stock pour les revendre	Détection des comportements automatisés, limites métier

#	Risque	Au bar, ça donne...	Se protéger
API7	Server Side Request Forgery (SSRF)	Une route « importer la photo du cocktail depuis une URL » à qui l'on donne <code>http://localhost:8080/admin</code>	Liste blanche de domaines, interdire les adresses internes
API8	Security Misconfiguration	Les stack traces en production, CORS ouvert à tous, HTTP sans TLS	Configuration revue, erreurs neutres, HTTPS partout
API9	Improper Inventory Management	L'API v0 du stagiaire de 2009, oubliée mais toujours en ligne	Inventaire des versions et des environnements, décommissionnement
API10	Unsafe Consumption of APIs	Faire aveuglément confiance aux données renvoyées par l'API du fournisseur de citrons	Valider aussi ce qu'on reçoit des API tierces

Les trois à retenir

Si vous ne deviez en retenir que trois, ce sont celles qui concernent l'**autorisation** : API1, API3 et API5. Elles ne sont détectées ni par les scanners automatiques, ni par les tests qui « vérifient que ça marche ». Il faut se demander, pour chaque route : *et si c'était quelqu'un d'autre qui appelait ?*

Le réflexe à prendre en écrivant une route : « Qui a le droit d'appeler cette route ? Sur quels objets ? Avec quels champs ? »

Référence complète : [OWASP API Security Top 10 - 2023](#).