Étape 2 — L'API PHP de réception

Mettre en place un endpoint PHP minimal qui reçoit le JSON, le valide, et le stocke proprement.

5TTR Exercice ambitieux

Objectif de l'étape

Construire le **côté serveur**. À la fin, ton script PowerShell pourra envoyer son rapport à `http://localhost/inventaire/api/report` au lieu de webhook.site, et le rapport sera **sauvegardé en JSON sur le disque**.

Pas de framework, pas de Composer, pas de base de données. Du PHP brut.

1. Structure du projet

Crée dans `C:\laragon\www\inventaire` (ou ton équivalent) cette arborescence :

```
inventaire/
├── public/                # racine web (à pointer dans Laragon)
│   ├── index.php          # routeur très simple
│   ├── api/
│   │   └── report.php     # endpoint POST de réception
│   ├── css/
│   └── js/
├── data/
│   └── pcs/               # créé automatiquement, un sous-dossier par PC
├── src/
│   ├── storage.php        # fonctions de lecture/écriture JSON
│   ├── validation.php     # fonctions de validation du payload
│   └── auth.php           # vide pour l'instant — utilisé à l'étape 6
└── README.md
```

> 💡 **POURQUOI `PUBLIC/` ?** Pour qu'on puisse plus tard servir uniquement ce dossier au web et garder `data/` et `src/` inaccessibles depuis l'extérieur. C'est une bonne habitude dès le départ.

2. Le routage minimal

Dans `public/index.php` :

```php
<?php
declare(strict_types=1);

require __DIR__ . '/../src/storage.php';
require __DIR__ . '/../src/validation.php';

// On découpe l'URL pour savoir ce qu'on doit afficher
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
$path = rtrim($path, '/');

// Très simple : on regarde le début du chemin
if (str_starts_with($path, '/inventaire/api/report')) {
    require __DIR__ . '/api/report.php';
    exit;
}

if ($path === '/inventaire' || $path === '/inventaire/') {
    require __DIR__ . '/dashboard.php';  // (sera créé à l'étape 4)
    exit;
}

if (str_starts_with($path, '/inventaire/pc/')) {
    require __DIR__ . '/pc.php';         // (sera créé à l'étape 3)
    exit;
}

http_response_code(404);
echo "404 – Page non trouvée";
```

> 💡 Si tu trouves ça compliqué, sache que **C'EST PLUS SIMPLE QU'UN FRAMEWORK**. Tu fais en 15 lignes ce que Symfony fait en 50 fichiers.

🤖 Travailler avec l'IA — routage

Tu peux demander à l'IA :

> Explique-moi `str_starts_with` en PHP, depuis quelle version c'est disponible, et donne-moi un équivalent compatible PHP 7 au cas où.

Pourquoi cette question ? Parce que **Laragon a peut-être PHP 8**, mais **l'IA va parfois te proposer du code PHP 5**. Tu dois savoir reconnaître le niveau de modernité du code qu'on te donne.

3. L'endpoint de réception — version minimale

Dans `public/api/report.php` :

```php
<?php
declare(strict_types=1);

// Toujours en JSON
header('Content-Type: application/json; charset=utf-8');

// On n'accepte que les POST
if ($_SERVER['REQUEST_METHOD'] !== 'POST') {
    http_response_code(405); // Method Not Allowed
    echo json_encode(['error' => 'Méthode non autorisée. Utilise POST.']);
    exit;
}

// Lecture du corps brut (PowerShell envoie du JSON, pas du form-url-encode
$raw = file_get_contents('php://input');

if ($raw === '' || $raw === false) {
    http_response_code(400);
    echo json_encode(['error' => 'Corps de requête vide.']);
    exit;
}

// Décodage
$data = json_decode($raw, true);
if (json_last_error() !== JSON_ERROR_NONE) {
    http_response_code(400);
    echo json_encode([
        'error' => 'JSON invalide.',
        'detail' => json_last_error_msg()
    ]);
    exit;
}

// À ce stade, $data est un tableau PHP. On va valider puis stocker.
// (Les fonctions sont dans src/)

$erreurs = valider_rapport($data);
```

```php
if (!empty($erreurs)) {
    http_response_code(422); // Unprocessable Entity
    echo json_encode(['error' => 'Données invalides.', 'detail' => $erreurs
    exit;
}

try {
    enregistrer_rapport($data);
} catch (Throwable $e) {
    http_response_code(500);
    echo json_encode(['error' => 'Erreur serveur.', 'detail' => $e->getMess
    exit;
}

http_response_code(201); // Created
echo json_encode([
    'ok'        => true,
    'hostname'  => $data['hostname'],
    'savedAt'   => date('c')
]);
```

⚠️ **POURQUOI `file_get_contents('php://input')` ET PAS `$_POST` ?** Parce que `$_POST` ne contient que les données envoyées en `application/x-www-form-urlencoded` (un formulaire HTML classique). Quand tu envoies du JSON, PHP ne le parse pas tout seul. Tu lis le flux brut, tu le décodes toi-même.

4. La validation

Dans `src/validation.php`. **Vraiment simple**, on vérifie juste que les champs essentiels existent :

```php
<?php

function valider_rapport(array $data): array
{
    $erreurs = [];

    if (empty($data['hostname']) || !is_string($data['hostname'])) {
        $erreurs[] = "Champ 'hostname' manquant ou invalide.";
    } elseif (!preg_match('/^[a-zA-Z0-9_\-]{1,64}$/', $data['hostname'])) {
        // Sécurité : on n'autorise QUE lettres, chiffres, _, -
```

```
        $erreurs[] = "Hostname contient des caractères interdits.";
    }

    if (empty($data['timestamp'])) {
        $erreurs[] = "Champ 'timestamp' manquant.";
    }

    foreach (['system', 'memory'] as $clef) {
        if (!isset($data[$clef]) || !is_array($data[$clef])) {
            $erreurs[] = "Section '$clef' manquante.";
        }
    }

    return $erreurs;
}
```

⚠️ **LE PIÈGE DE SÉCURITÉ MAJEUR** : le hostname va devenir un **nom de dossier**. Si un attaquant envoie `hostname = "../../../etc/passwd"` , il peut écrire **n'importe où** sur ton disque. La regex ci-dessus empêche ça. **Ne supprime jamais cette ligne.**

5. Le stockage

Dans `src/storage.php` :

```php
<?php

const DATA_ROOT = __DIR__ . '/../data/pcs';

function enregistrer_rapport(array $data): void
{
    $hostname = $data['hostname'];
    $dossier  = DATA_ROOT . '/' . $hostname;

    // Crée le dossier (et l'historique) si nécessaire
    if (!is_dir($dossier . '/history')) {
        mkdir($dossier . '/history', 0775, true);
    }

    // 1) Snapshot horodaté dans l'historique
    $timestamp = date('Ymd-His'); // ex : 20260513-103015
```

```php
        $fichierHistorique = $dossier . "/history/$timestamp.json";
        file_put_contents(
            $fichierHistorique,
            json_encode($data, JSON_PRETTY_PRINT | JSON_UNESCAPED_UNICODE)
        );

        // 2) Latest = dernière version connue (écrasée à chaque fois)
        $fichierLatest = $dossier . '/latest.json';
        file_put_contents(
            $fichierLatest,
            json_encode($data, JSON_PRETTY_PRINT | JSON_UNESCAPED_UNICODE)
        );
    }
```

> 💡 **POURQUOI DEUX FICHIERS ?** Le `latest.json` est lu **très souvent** (dashboard, fiche PC). On veut un accès ultra-rapide à la dernière valeur. L'historique sert au graphe d'évolution. C'est un compromis classique entre **rapidité de lecture** et **conservation des données**.

🤖 Travailler avec l'IA — discussions de design

Tu peux demander :

> Pourquoi stocker à la fois un `latest.json` et un dossier `history/` ? Quel est le compromis ? Quand est-ce que ce serait une mauvaise idée ?

C'est une **bonne** question à poser à l'IA, parce qu'elle t'explique des **alternatives** (BDD, log unique append-only, fichier par jour…) sans te livrer du code. Tu apprends à raisonner en architecte.

Mauvaise question :

> Refais mon stockage en utilisant SQLite à la place.

Trop tôt, et hors-sujet pédagogique. Reste sur le fichier.

6. Tester l'API

Tu as **trois manières** de tester ; utilise les trois, dans cet ordre :

a) En ligne de commande, avec PowerShell

```powershell
$body = @{ hostname = "TEST"; timestamp = (Get-Date).ToString("o");
           system = @{}; memory = @{} } | ConvertTo-Json

Invoke-RestMethod -Uri "http://localhost/inventaire/api/report" `
                  -Method Post `
                  -ContentType "application/json" `
                  -Body $body
```

Tu dois recevoir `{ ok = True; hostname = TEST }`.

b) Avec Thunder Client (extension VS Code)

Encore plus pratique : tu peux **enregistrer** la requête, la rejouer, et voir le code de statut.

c) En modifiant volontairement le payload

Envoie successivement :

1. Un POST vide → tu dois obtenir un **400**.
2. Un POST avec un JSON cassé → **400** avec `detail`.
3. Un POST sans `hostname` → **422**.
4. Un POST avec `hostname = "../etc"` → **422** (validation refuse).

Si une de ces 4 attaques aboutit à un 200, tu as un bug à corriger.

7. Brancher PowerShell sur ton API

Modifie le `-ApiUrl` de ton script de l'étape 1 :

```powershell
.\inventaire.ps1 -ApiUrl "http://localhost/inventaire/api/report"
```

Le serveur PHP doit te répondre `{ ok: true, hostname: "...", savedAt: "..." }`, et tu dois trouver un fichier dans `C:\laragon\www\inventaire\data\pcs\TON-PC\latest.json`.

Ouvre ce fichier et vérifie qu'il est lisible, bien indenté, et que les accents passent bien.

✅ Critères de réussite de l'étape 2

- [ ] L'arborescence est conforme.
- [ ] Un POST vide / mal formé renvoie un **400** ou **422**, jamais 500.

- [ ] Un POST correct renvoie un **201** et crée bien les fichiers.
- [ ] Le fichier `latest.json` est écrit en UTF-8, lisible.
- [ ] L'historique se remplit à chaque envoi.
- [ ] Tu peux expliquer **pourquoi** on lit `php://input` au lieu de `$_POST`.

⚠️ Récap des pièges

Piège	Solution
Encodage cassé côté PowerShell	`-Encoding utf8` en sortie, `charset=utf-8` en header
`$_POST` vide alors qu'on envoie du JSON	Lire `php://input`
`ConvertTo-Json` perd les sous-objets	`-Depth 6` minimum
Hostname `..\..\..` qui écrit n'importe où	Regex de validation stricte
Dossier `data/` accessible depuis le web	Mettre `data/` **hors** de `public/`

Pour la suite

À l'étape 3, on lit ce JSON et on l'affiche dans une **belle fiche PC**.