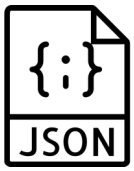
Kenzo MARIE

Module de connexion PrestaShop



Sommaire

I. EXPRESSION DES BESOINS

1.1. Contexte du projet

Dans le cadre de ma formation en tant qu'étudiant de deuxième année en BTS SIO, j'ai réalisé mon stage chez la société Ether Création, société française spécialisé dans le développement informatique, plus précisément, dans la conception et la réalisation de modules sur PrestaShop.

La société m'a assigné comme mission de **concevoir et réaliser un module/connecteur ou ECI (External Call Inferface,** interface d'appel externe) sur PrestaShop, ainsi que comprendre comment fonctionne la communication entre celui-ci et une **ERP** (**Enterprise Resource Planning,** planification des ressources d'entreprise,)

1.2. Acteurs et parties prenantes

- **Les Administrateurs (Back-Office) :** Le module leur offre une interface de suivi des flux, leur évitant la double saisie manuelle des données, source d'erreurs et de perte de temps.
- **Les Clients finaux (Front-Office) :** Ils bénéficient d'une mise à jour en temps réel des stocks et des prix, garantissant la fiabilité des commandes.

1.3. Étude de l'existant

Avant mon intervention, un précédent **ECI** avait été créé pour des besoins spécifiques, j'ai donc repris la base de ce module, qui n'avait pas d'écriture liée à une API, afin de le modifier pour qu'il ait des fonctionnalités similaire mais en lien avec l'API recherchée, et qu'il puisse agir sur un étendu de tâches à faire contenues dans PrestaShop.

1.4. Cadrage de la demande

La demande consistait à développer un module capable de :

- Lire et interpréter des flux de données (CSV) provenant de l'ERP.
- Organiser et trier ces flux (traitement des données brutes vers données utiles).
- S'intégrer nativement dans le Back-Office de PrestaShop pour le monitoring.

1.5. Contraintes et exigences

Il m'a été demandé de concevoir et réaliser un module en utilisant la technologie Programmation Orientée Objet en langage PHP, celui-ci fait la liaison entre PrestaShop natif et une ERP. Avant de commencer sa réalisation, il a été nécessaire d'effectuer des recherches approfondis sur comment parvenir à la réalisation du module demandé, et de participer activement à la vie de l'entreprise afin d'avoir un aperçu professionnel de la tâche demandée.

II. CONCEPTION ET SPÉCIFICATIONS TECHNIQUES

2.1. Présentation de la solution

1. Présentation générale

Pour répondre au besoin de synchronisation entre l'ERP et le site E-commerce, j'ai développé une application intermédiaire nommée **ECI (External Call Interface)**. Cette solution permet de traduire les formats de données de l'ERP vers PrestaShop et inversement.

2. Architecture technique

L'architecture repose sur un modèle d'échange de fichiers asynchrone.

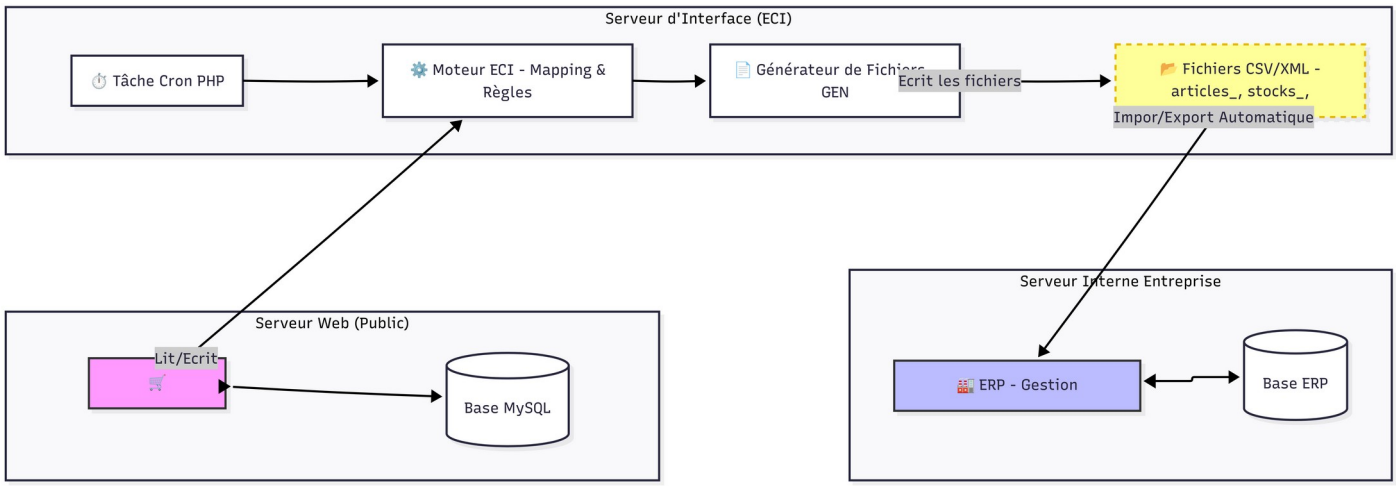
- **Côté Serveur :** Le script est hébergé sur le serveur Web.
- **Automatisation :** Une tâche **CRON** (`$cronstate` dans le code) déclenche les traitements à intervalles réguliers.
- **Flux de données :** Comme le montre le **Schéma**, les données transitent via des fichiers CSV identifiés par des masques (ex: `articles_`, `stocks_`).

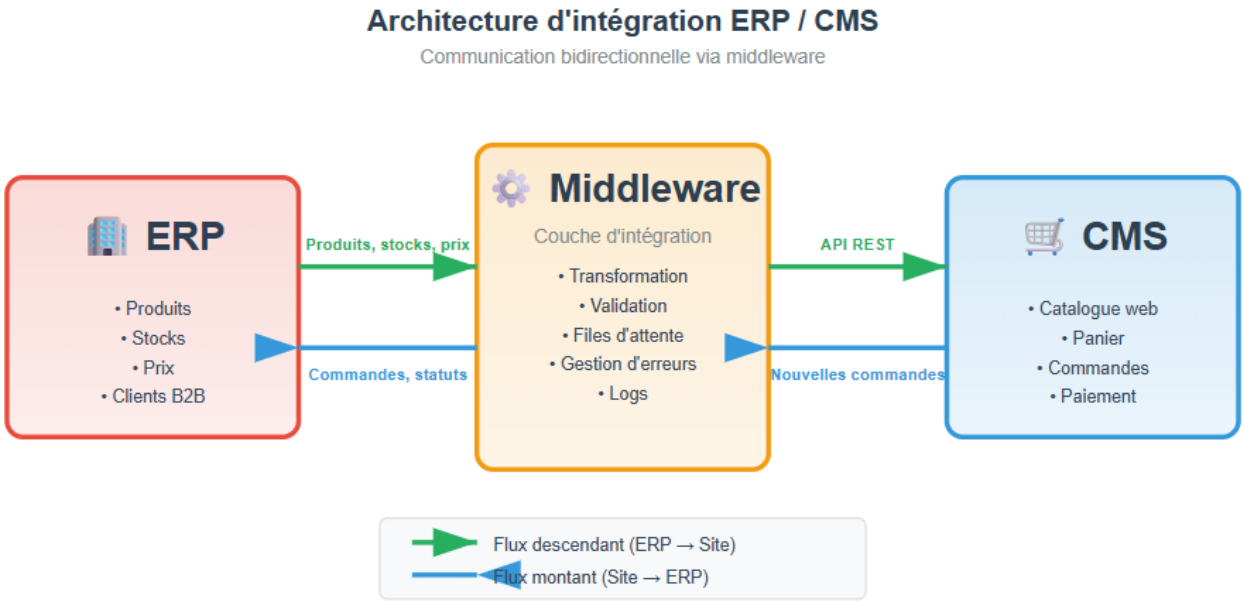
[Voir schéma de l'architecture ci-dessous]

3. Choix de conception

J'ai opté pour une programmation Orientée Objet en PHP pour faciliter la maintenance. Le système est modulaire : chaque type de flux (Catalogue, Stock, Client) est géré indépendamment, ce qui permet d'ajouter de nouveaux flux (comme `test` ou `tracking`) sans casser l'existant.

2.2. Architecture technique





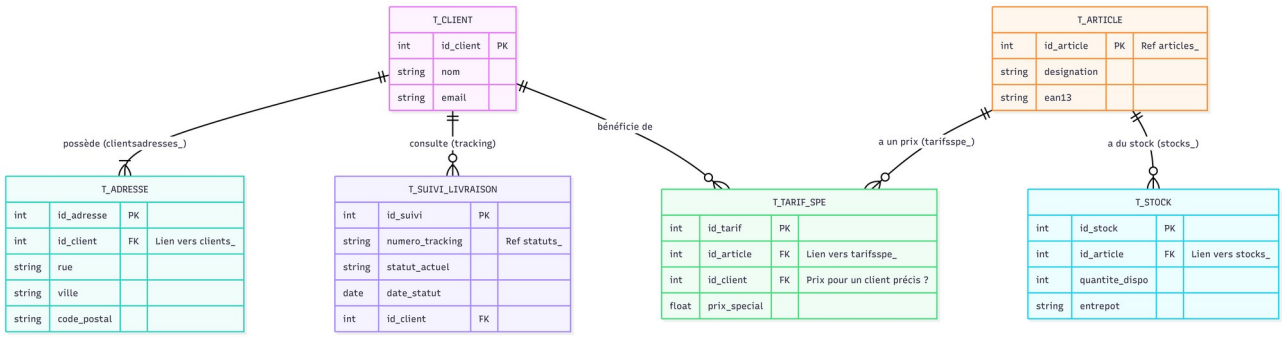
2.3. Ressources et prérequis

Le projet nécessite l'utilisation d'un IDE (VSCode), d'un environnement serveur LAMP (Linux Apache MySQL PHP), ainsi que l'utilisation d'un logiciel appelé PostMan pour effectuer des requêtes HTTP. Aucun coût de licence supplémentaire n'a été engendré, les technologies utilisées étant Open Source ou bien gratuites.

2.4. Modélisation et analyse des données

- Séparation stricte des entités `Clients` et `Adresses` pour gérer le multi-adresses.

- Tables dédiées pour les stocks déportés et les tarifications spécifiques par client.



2.5. Pilotage et conduite de projet

Notre projet est découpé en plusieurs phases, chacune d'entre elles est liée à une tâche (voir capture ci-dessus) bien précise, par exemple : la tâche « Rafraîchissement du catalogue » qui, comme son nom l'indique, sert à rafraîchir les données envoyées pour remplir le catalogue.

Dans notre cas, notre tâche principale est ciblé sur l'apparence et la composition du catalogue.

Dans un cadre où plusieurs tâches sont nécessaires, la conduite de projet est divisée en quatre grandes parties :

-**Premièrement**, l'étude de l'existant, et l'étude du besoin du client

-**Deuxièmement**, la phase de mise en production, écrire les fonctions nécessaires, en suivant le besoin du client.

-**Troisièmement**, la phase de test et de retour de la part du client, afin d'avoir la possibilité de modifier les fonctionnalités si elles ne conviennent pas aux attentes du client.

-**Dernièrement**, La mise en production finale avec rendu du projet complet et fonctionnel.

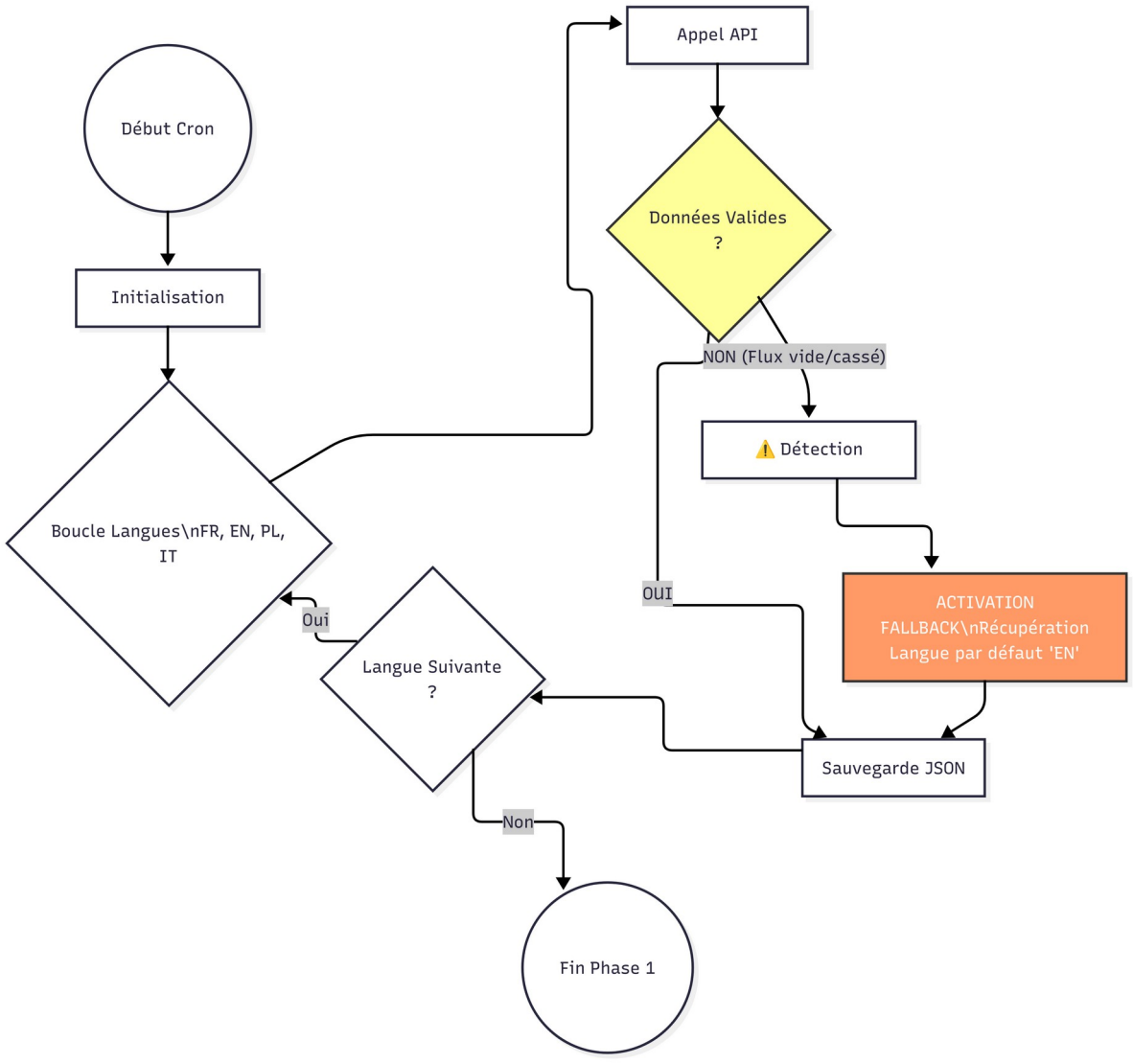
III. DÉVELOPPEMENT ET RÉALISATION

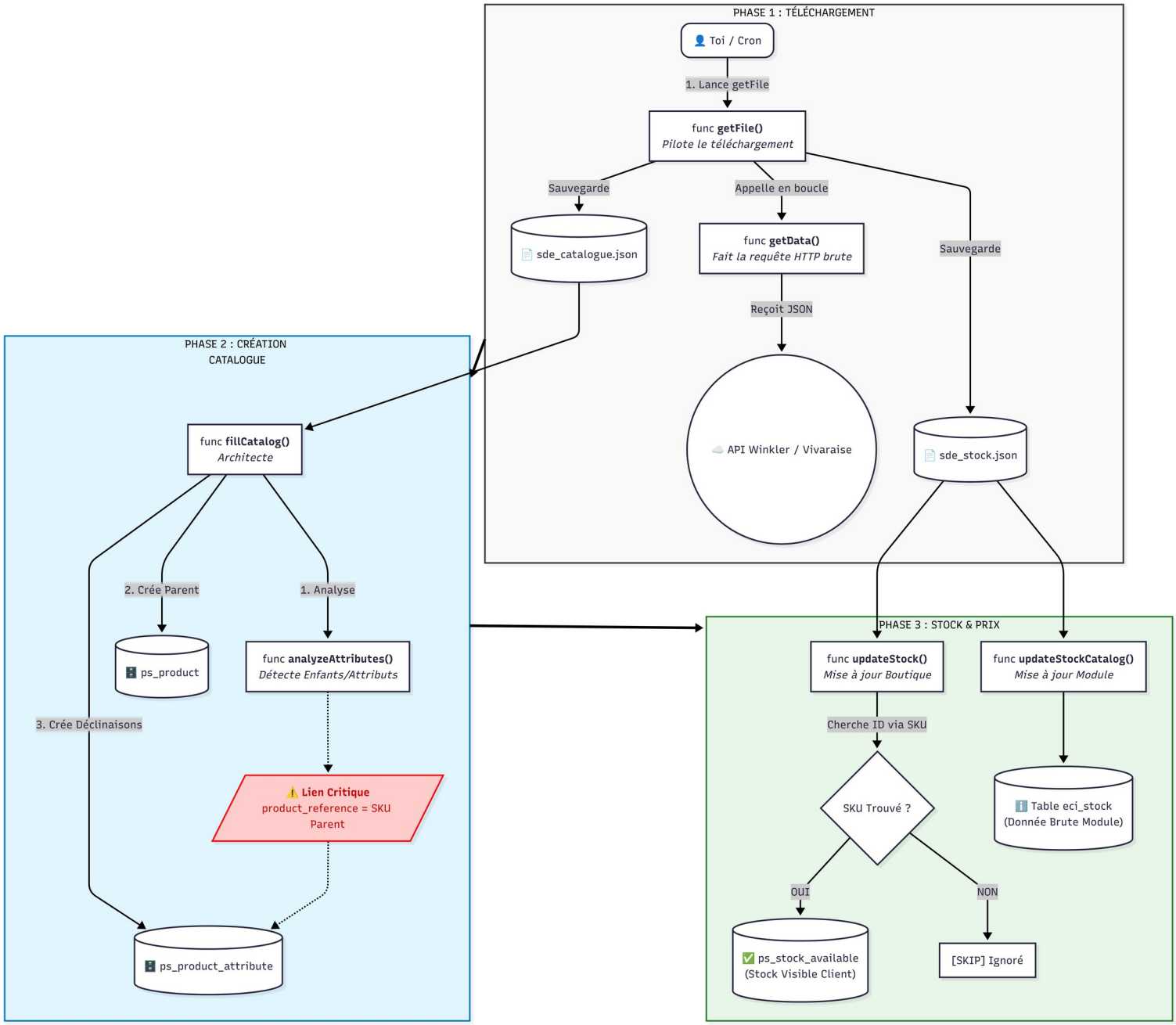
3.1. Implémentation de la solution

Le développement s'est concentré sur la création d'un ECI (connecteur/module) sur une boutique PrestaShop. L'objectif était de concevoir des fonctions en lien avec une succession de tâches, comme par exemple : l'affichage du catalogue, après rafraîchissement des données reçues de l'API distante.

3.2. Dossier de programmation codes sources documentés et commentés

Le premier défi était de gérer un flux API. Si je demande le catalogue en Polonais et que l'API me renvoie un tableau de données vide, je ne veux pas écraser mon catalogue PrestaShop avec du vide. J'ai donc mis en place un système de **Failover** (bascule) automatique. Afin que cette possibilité soit prise en compte, et donc traiter l'intégrité efficacement. Dans ce contexte, si une langue n'existe pas, elle est remplacée par la langue par défaut instauré sur PrestaShop (le français ici).





`getFile` importe la matière première (les fichiers JSON du fournisseur),

`fillCatalog` utilise ces données pour construire la structure des produits et leurs déclinaisons, et enfin

`updateStock` remplit les rayons en appliquant les prix et les quantités pour rendre le tout vendable.

```php
$fluxfinal = [];

foreach(array_keys($work) as $flux){
    $fluxConfig = $work[$flux];
    $debugLogs[] = "L" . __LINE__ . ": Analyse du flux : " . $flux;
    if(isset($fluxConfig['multilang']) && $fluxConfig['multilang'] === true) {
        $languesDetectees = array_keys(Catalog::getAllLanguages());
        $debugLogs[] = "DEBUG : Langues trouvées dans PrestaShop : " . implode(', ', $languesDetectees);

        foreach ($languesDetectees as $iso) {
            $fluxlang = $flux . '_' . $iso;
            $newflux = $fluxConfig;
            $newflux['lang'] = $iso;
            $newflux['multilang'] = false;
            $fluxfinal[$fluxlang] = $newflux;
        }
    } else {
        $fluxfinal[$flux] = $fluxConfig;
    }
}
$work = $fluxfinal;

$filesToDo = implode(',', array_keys($work));
$debugLogs[] = "L" . __LINE__ . ": Liste des fichiers prévus : " . $filesToDo;

if ($langIso === 'fr') {
    $defaultLang = 'fr';
} else {
    $defaultLang = 'en';
}
```

Cet extrait de code illustre la **gestion dynamique du multilinguisme**. Le script détecte automatiquement les langues actives sur la boutique PrestaShop et démultiplie les requêtes API pour chaque marché (FR, EN, IT, etc.). Cette logique est fondamentale pour une boutique internationale : elle garantit l'importation des descriptions et titres dans la bonne langue pour chaque déclinaison du catalogue, assurant ainsi une expérience client native et un référencement (SEO) optimal. Et, dans le cas où la langue par défaut de la boutique n’est pas définie au ‘français’, celle-ci sera directement définie à l’anglais, cela fortifie l’intégrité des données car certaines langues ne sont pas présentes sur PrestaShop (exemple : le polonais).

3.3. Retour d'expérience technique

Difficultés rencontrées, gestion des bugs et fonctionnalités restantes (Reste à faire)

La première difficulté à laquelle j'ai été exposé très tôt, c'était le flux de données en multilangues, donc un seul flux contenant plusieurs langues (exemple : anglais, français et italien). Je devais récupérer ce flux contenant toutes les langues, et le démultiplier en plusieurs flux distincts les uns des autres, ainsi, chaque flux est unique et ne contient qu'une seule langue. (un flux anglais, un flux français, …). Exemple ci-dessous :

```php
foreach(array_keys($work) as $flux){
    $fluxConfig = $work[$flux];
    $debugLogs[] = "L" . __LINE__ . ": Analyse du flux : " . $flux;
    if(isset($fluxConfig['multilang']) && $fluxConfig['multilang'] === true) {
        $languesDetectees = array_keys(Catalog::getAllLanguages());
        $debugLogs[] = "DEBUG : Langues trouvées dans PrestaShop : " . implode(', ', $languesDetectees);

        foreach ($languesDetectees as $iso) {
            $fluxlang = $flux . '_' . $iso;
            $newflux = $fluxConfig;
            $newflux['lang'] = $iso;
            $newflux['multilang'] = false;

            $fluxfinal[$fluxlang] = $newflux;
        }
    } else {
        $fluxfinal[$flux] = $fluxConfig;
    }
}
$work = $fluxfinal;
```

Une seconde difficulté à laquelle j'ai été exposé, traité les produits qui n'ont qu'une seule déclinaison. Ce que je n'avais pas prévu à l'origine dans ma fonction 'analyzeAttributes', qui me permettait de comparer les valeurs afin de dénicher les attributs, c'était qu'un produit référent ne puisse avoir qu'une seule déclinaison, donc ma comparaison était impossible. J'ai donc pris en compte cette option ensuite, en forçant le fait que si une déclinaison avait un attribut (exemple : couleur), c'était son attribut, pas besoin de comparaison.

```php
if ($nbChild === 1) {
        foreach ($this->tab_attr as $name => $tag) {
            if (!empty($tag)) {
                $Attributes[$name] = $tag;
            }
        }
        return [
            'attributes' => $Attributes,
            'features' => []
        ];
}
```

IV. EXPLOITATION ET MISE EN PRODUCTION

4.1. Stratégie de Tests

Compétence 5.3

Avant la mise en production, une batterie de tests a été effectuée sur un environnement de "Pré-production":

- **Tests unitaires :** Vérification de chaque fonction du code, mise en place de formule de debug, de suivi des lignes, et de messages ressortant les possibles erreurs.

```
{
    "user": "{{eciapilogin}}",
    "authtoken": "{{eciapipass}}",
    "query": {
        "action": "ecsde::apiCall",
        "parameters": {
            "function": "fillCatalog",
            "id_shop": 1,
            "params": [
                "catalog",
                "",
                0
            ]
        }
    }
}
```

Voici une des requêtes de test faites sur PostMan, celle-ci est dirigée vers « fillCatalog », une fonction écrite sur le code du module. (fillCatalog sert à rafraîchir et remplir le catalogue PrestaShop avec les produits contenus dans le flux).

- **Tests d'intégration :** Simulation d'une commande complète et vérification de sa remontée vers l'ERP.

Les deux captures ci-contre montrent une requête **HTTP POST** qui interroge l'**API** grâce aux fonctions préalablement développées sur notre **ECI** *(capture d'écrans dans la partie développement)* Afin que celle-ci nous autorise à collecter les données et les interpréter ensuite dans le format choisi, en l'occurrence ici : en **RAW**(brute) **JSON.**

```
{
    "user": "{{eciapilogin}}",
    "authtoken": "{{eciapipass}}",
    "query": {
        "action": "ecsde::apiCall",
        "parameters": {
            "function": "getData",
            "id_shop": 1,
            "params": [
                "products",
                {
                    "lang": "fr"
                },
                "POST"
            ]
        }
    }
}
```

V. BILAN DU STAGE