Présentation Générale de PokeSync

Documentation Technique – PokeSync

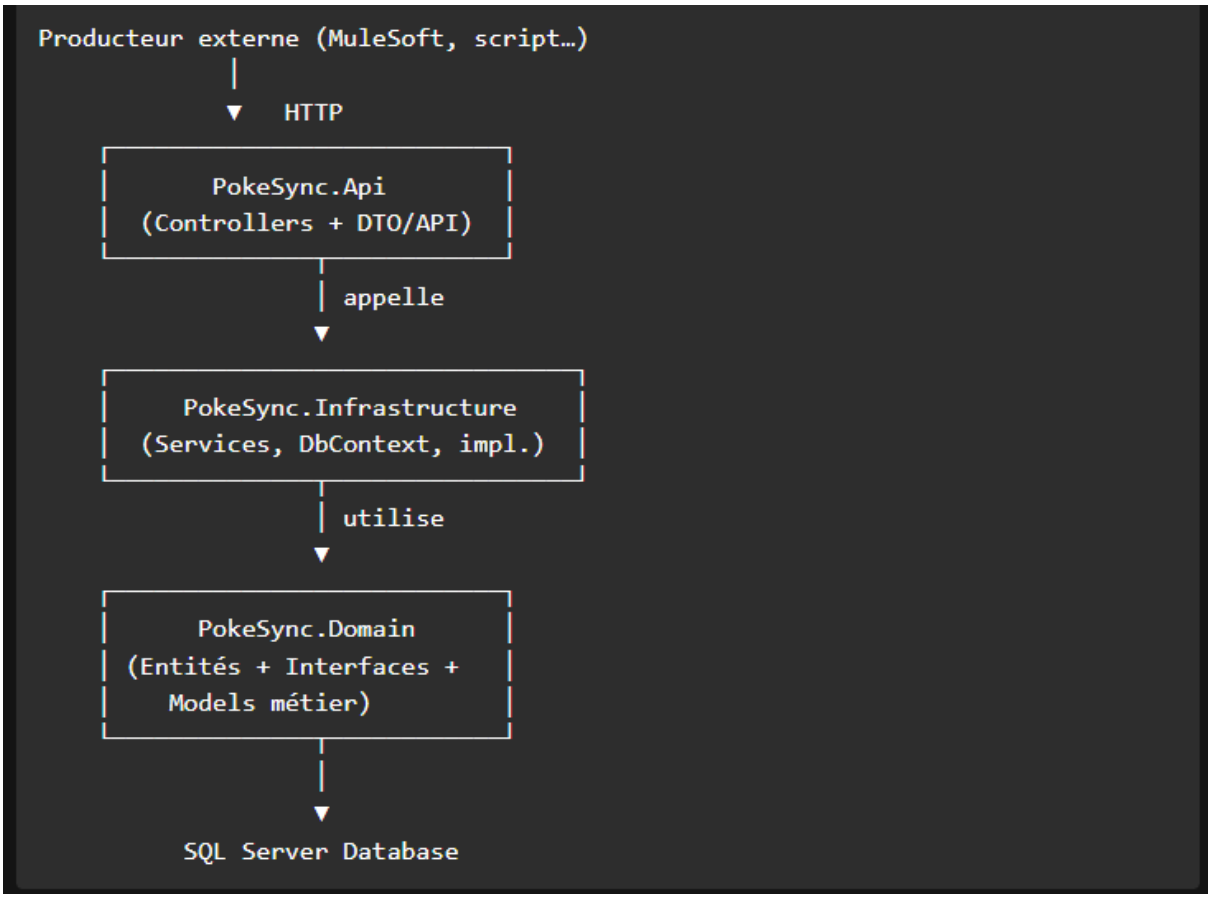
1. Présentation Générale de PokeSync

PokeSync est une solution backend développée en .NET visant à centraliser, structurer et synchroniser des données Pokémon provenant d'un producteur externe (par exemple MuleSoft ou un script autonome). Le but de la solution est d'offrir une base technique propre, évolutive et fiable, construite selon une architecture claire et professionnelle.

🎯 Objectifs du projet (version actuelle)

- **Recevoir un batch de données Pokémon via un endpoint interne.**
- **Réaliser un upsert en masse (insertion + mise à jour intelligente) en base SQL Server.**
- **Garantir la cohérence et l'idempotence des synchronisations successives.**
- **Exposer un endpoint public permettant de vérifier l'état du système (initialisation, dernière synchronisation, etc.).**
- **Fournir une architecture propre qui pourra être enrichie (UI, nouvelles sources, automatisation, etc.).**

🧩 Architecture générale (périmètre actuel)



🏗️ Une base saine pour la suite

Cette architecture a été pensée pour rester simple, lisible et extensible.
Elle permet aujourd'hui :

- **de synchroniser proprement les données,**
- **de garantir leur intégrité,**
- **de disposer d'un socle professionnel pour des évolutions futures.**

Principes d’Architecture & Choix Techniques

2. Principes d’Architecture & Choix Techniques

L’architecture de PokeSync repose sur des principes professionnels inspirés de la Clean Architecture, mais adaptés de manière **pragmatique** au périmètre fonctionnel actuel.

Le résultat : une structure **simple, claire, modulaire** et aisément extensible.

2.1 Objectifs architecturaux

Les choix techniques visent à :

- **Séparer nettement les responsabilités** pour obtenir un code lisible et testable.
- **Isoler la logique métier** dans un espace qui ne dépend d’aucune technologie.
- **Centraliser les règles de synchronisation** et de persistance dans un espace dédié.
- **Limiter la logique dans l’API**, afin qu’elle reste une simple façade d’exposition.
- **Garantir la stabilité dans le temps**, en permettant d’ajouter de nouvelles fonctionnalités sans casser l’existant.

2.2 Les trois couches de la solution

PokeSync est structuré en trois projets principaux, chacun ayant un rôle clairement défini.

```
PokeSync.Domain         →  Règles métier
PokeSync.Infrastructure →  Logique applicative + accès aux données
PokeSync.Api            →  Exposition HTTP
```

🔹 PokeSync.Domain — Le cœur stable

Contient :

- les **entités métier** (Pokemon, ElementType, Generation, etc.),
- les **interfaces** des services (contrats fonctionnels),
- les **models métier** utilisés pour l’upsert, les résultats, etc.

Pourquoi ?
Parce que le Domain est la **base de vérité fonctionnelle**.
Il ne dépend d'aucune technologie : ni EF Core, ni ASP.NET, ni SQL Server.

Cela permet de :

- tester facilement la logique métier,
- réutiliser le Domain dans d'autres applications si nécessaire,
- garder les règles métiers indépendantes des détails techniques.

🔹 PokeSync.Infrastructure — Le moteur de l'application

Regroupe :

- le `PokeSyncDbContext` et les **migrations EF Core**,
- l'**implémentation** de toutes les interfaces du Domain,
- la **logique applicative** (upsert, idempotence, statut système),
- les accès à la base SQL Server.

Pourquoi ?
L'Infrastructure a la responsabilité de :

- transformer les intentions métier en opérations réelles,
- gérer la résistance aux erreurs,
- orchestrer l'écriture et la lecture dans la base.

C'est la couche qui « travaille », tandis que le Domain définit quoi faire.

🔹 PokeSync.Api — La façade HTTP

Contient :

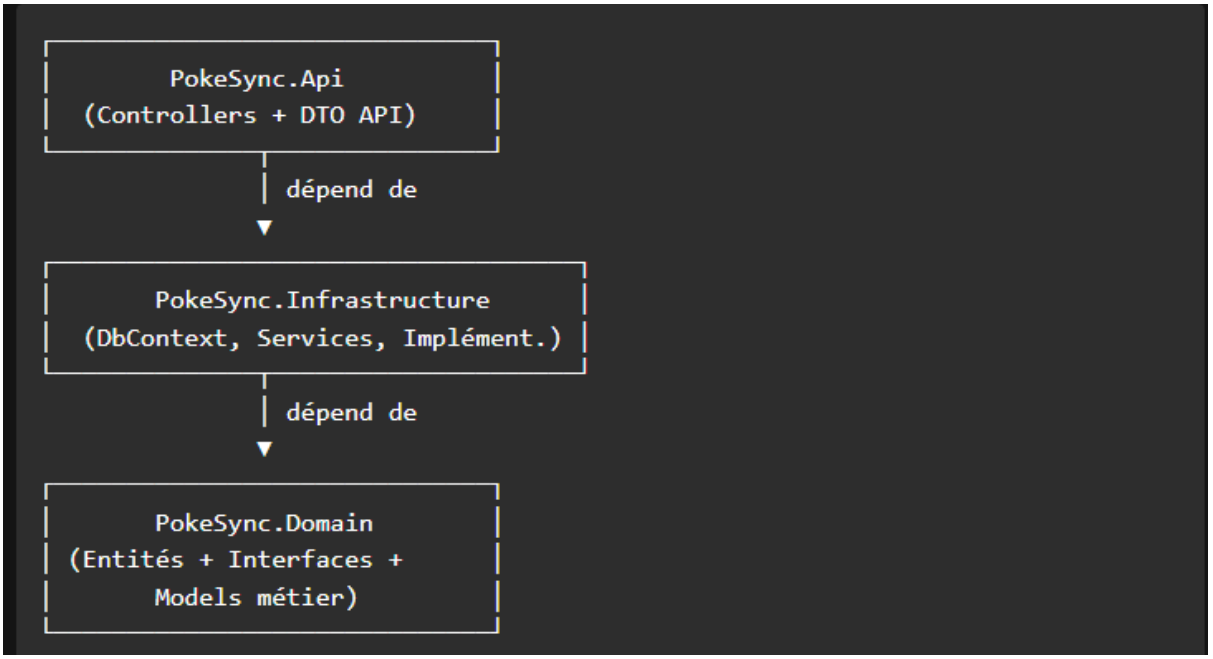
- les **Controllers**,
- les **DTO d’entrée et de sortie**,
- la configuration de l’application.

Pourquoi ?
Cette couche doit rester **minimale** :

- elle reçoit une requête,
- valide les données au minimum,
- appelle le service nécessaire,
- renvoie la réponse HTTP.

Toute la logique métier et applicative est gérée par Domain + Infrastructure.

2.3 Schéma général de dépendances



Ce schéma illustre une règle fondamentale :
👉 **la dépendance va toujours du plus externe vers le plus interne**.

- L'API dépend d'Infrastructure.
- Infrastructure dépend de Domain.
- Domain ne dépend de rien.

C'est une architecture propre, stable, et facile à faire évoluer.

NB : Pour un projet *simple* (comme une API d'intégration), avoir une couche *Application* séparée de la couche *Domaine* ajoute souvent du *boilerplate* (code répétitif) sans réel gain de complexité.

2.4 Choix techniques clés

✔ EF Core + Migrations

Permet une base de données versionnée, traçable et reproductible.

✔ Interfaces dans Domain

Le Domain définit **ce que le système doit faire**, sans se soucier du « comment ».

L'Infrastructure fournit ensuite le « comment ».

✔ Modèles métier dans Domain

Les `PokemonUpsertItem`, `UpsertBatchResult`, etc. étant indépendants de la persistance, il est naturel qu'ils vivent dans le Domain.

✔ Idempotence

Mise en place via :

- une entité dédiée (`IdempotencyKey`),
- un service dédié (`IdempotencyService`),
- un mécanisme de stockage d'état.

Cela évite :

- les doublons,
- les réinjections inutiles,
- les corruptions lors de synchronisations multiples.

✔ API propre et minimale

Les Controllers restent courts :

- pas de logique métier,
- pas d'accès direct à la base,
- simplement un mapping + appel de service.

Cela garantit un code clair et simple à maintenir.

2.5 Résultat : une architecture propre, simple et solide

Cette structure offre une séparation nette entre :

- **métier** (Domain),
- **mise en œuvre** (Infrastructure),
- **exposition** (Api).

Elle prépare le terrain pour :

- l’ajout d’autres APIs,
- une interface Angular,
- d’autres sources de synchronisation,
- des tests unitaires ciblés.

Une base saine, propre et professionnelle.

Domain

3. PokeSync.Domain

Le projet **PokeSync.Domain** constitue le cœur fonctionnel de la solution.
Il regroupe tout ce qui doit rester **stable**, **indépendant de la technologie**, et **représentatif du métier**.

Il ne référence aucune autre couche : ni Infrastructure, ni Api.
Au contraire, **Infrastructure et Api dépendent de lui**.

Ce projet contient trois catégories essentielles :

- les **entités métier**,
- les **interfaces** (contrats),
- les **models métier** utilisés lors des opérations (notamment l'upsert).

3.1 Rôle de la couche Domain

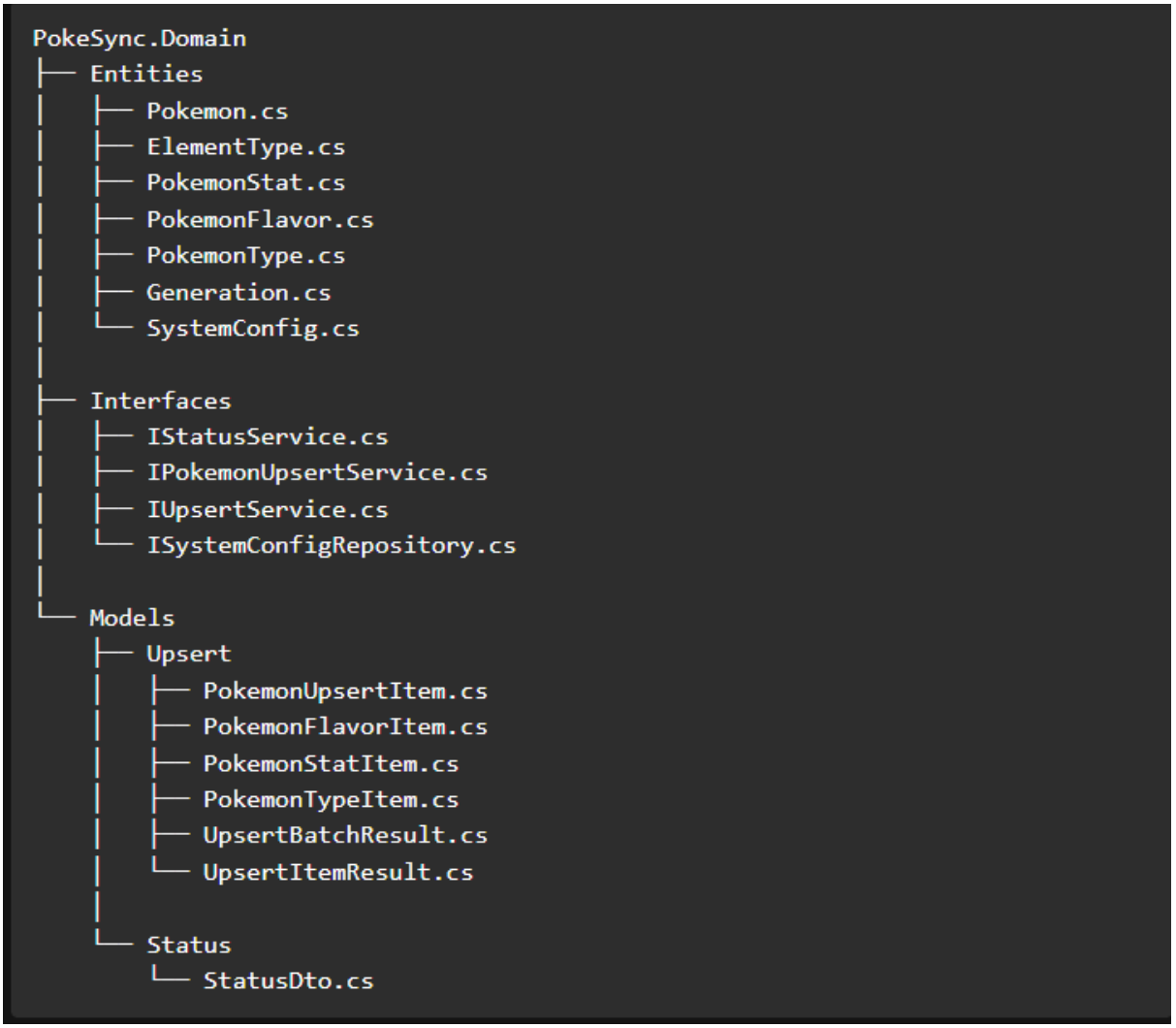
Le Domain définit :

- ce qu'est un Pokémon,
- comment les types, générations, stats ou descriptions sont structurés,
- comment les services doivent se comporter (interfaces),
- comment les données doivent être reçues lors d'une synchronisation (models d'upsert),
- comment un résultat d'upsert doit être exprimé (models de résultat),
- l'état système minimal (ex : initialisation, dernière synchro).

Le Domain **ne réalise aucune opération** : il décrit **le contrat** et **les structures**, pas leur exécution.

3.2 Structure du projet Domain

Organisation simplifiée du dossier :



Cette structuration rend le Domain clair, cohérent et facilement extensible.

3.3 Entités métier

Les entités représentent la **structure persistée** et le **modèle métier central**. Elles ne contiennent pas de logique, seulement des données cohérentes.

🔹 Pokémon

Représente un Pokémon complet dans la base.

Composition :

- identifiant Pokédex,
- nom,
- génération,
- stats,
- types,
- descriptions (flavors).

🔹 ElementType

Représente les types élémentaires (Feu, Eau, Plante…).

🔹 PokemonStat

Représente une statistique (PV, Attaque, Défense…).

🔹 PokemonFlavor

Contient une description textuelle (flavor text).

🔹 PokemonType

Association Pokémon ↔ Type.

🔹 Generation

Stocke les générations (Gen 1 à Gen X).

🔹 SystemConfig

Stocke l'état système minimal :

- Initializing (bool)
- LastSyncUtc (DateTime?)

Cette entité est utilisée pour le suivi global.

3.4 Interfaces

Les interfaces définissent **ce que la solution doit faire**, sans dire **comment**. Elles sont implémentées dans Infrastructure.

🔹 IStatusService

Expose la lecture de l'état du système.

🔹 IPokemonUpsertService

Définit les opérations d'upsert d'un ensemble de Pokémon.

🔹 IUpsertService

Service générique chargé d'orchestrer les opérations d'upsert.

🔹 ISystemConfigRepository

Accès abstrait aux données de configuration système.

Ces interfaces permettent :

- d'injecter dynamiquement des implémentations,
- d'éviter les dépendances technologiques dans le Domain,
- de faciliter les tests unitaires (mocking).

3.5 Models métier (Upsert & Statut)

Ces models représentent **les données manipulées** lors d'opérations métier. Ils ne sont pas liés directement à la BD : ce ne sont pas des entités persistées.

🔹 PokémonUpsertItem

Structure d'un Pokémon reçu lors d'un batch :

- PokedexId,
- Name,
- Types,
- Stats,
- Flavors,
- Generation, …

🔹 StatItems, FlavorItems, TypeItems

Déclinaisons pour les différents sous-objets.

🔹 UpsertBatchResult & UpsertItemResult

Expriment le résultat d'une synchronisation.

Exemples :

- combien d'entrées ont été créées,
- combien ont été mises à jour,
- combien ont échoué,
- détails des erreurs éventuelles.

🔹 StatusDto

Représente le statut système :

- Initializing,
- LastSyncUtc.

Ce model sert d'interface entre les services et l'API.

3.6 Rôle stratégique du Domain

Le Domain est volontairement **technologie-agnostique**, ce qui permet :

- d'écrire des tests unitaires uniquement sur la logique métier,
- de remplacer SQL Server par un autre stockage sans toucher au cœur métier,
- de remplacer l'API par une autre exposition (gRPC, CLI, etc.),
- d'étendre les fonctionnalités sans casser la base du projet.

Le Domain est le **contrat** et la **voix métier** de PokeSync.
Tout le reste n'est que mise en œuvre.

Infrastructure

4. PokeSync.Infrastructure

Le projet **PokeSync.Infrastructure** est la couche qui « met les mains dans le cambouis ». C'est elle qui :

- implémente les **interfaces** définies dans le Domain,
- orchestre la **logique applicative** (upsert, statut, idempotence),
- accède à la **base de données SQL Server** via EF Core,
- applique les **migrations** et le mapping entre entités et tables.

Elle dépend de `PokeSync.Domain`, mais n'est référencée ni par le Domain, ni par la base de code externe : tout passe par les interfaces.

4.1 Rôle de la couche Infrastructure

Concrètement, cette couche :

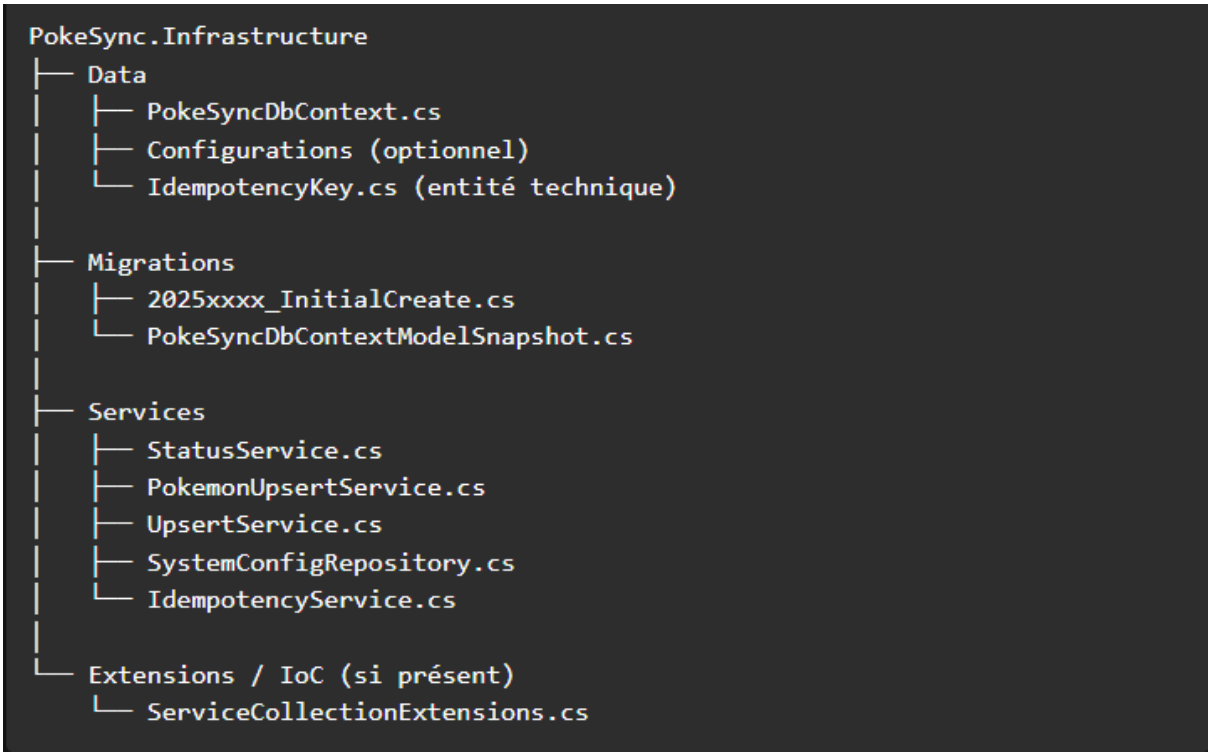
- traduit les intentions métier (interfaces du Domain) en **opérations concrètes** sur les données,
- encapsule EF Core et le `DbContext` pour éviter que le reste de l'application ne manipule directement la persistence,
- centralise la logique métier « opérationnelle » (idempotence, calcul des résultats d'upsert, agrégation des données),
- fournit des services injectables dans l'API.

L'objectif est que :

- l'API ne connaisse que les **interfaces**,
- le Domain ne connaisse pas EF Core,
- la base soit **remplaçable** sans casser toute l'application.

4.2 Structure du projet Infrastructure

Organisation simplifiée :



4.3 PokeSyncDbContext & accès aux données

Le `PokeSyncDbContext` est la porte d'entrée vers la base SQL Server.
Il expose des `DbSet<TEntity>` pour les entités du Domain et, éventuellement, des entités purement techniques.

Exemples typiques :

```
public class PokeSyncDbContext : DbContext
{
    public DbSet<Pokemon> Pokemons => Set<Pokemon>();
    public DbSet<ElementType> ElementTypes => Set<ElementType>();
    public DbSet<Generation> Generations => Set<Generation>();
    public DbSet<SystemConfig> SystemConfigs => Set<SystemConfig>();

    public DbSet<IdempotencyKey> IdempotencyKeys => Set<IdempotencyKey>();

    public PokeSyncDbContext(DbContextOptions<PokeSyncDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
        // Configuration fine des entités, clés composites, relations, etc.
    }
}
```

Les migrations EF Core définissent le schéma de base initial (tables, clés, relations) et permettent d'évoluer sans perte de contrôle.

4.4 Services et implémentation des interfaces

Les services de cette couche implémentent les interfaces du Domain et regroupent la **logique applicative**.

🔹 StatusService

Implémente `IStatusService`.

Rôle :

- lire la configuration système (`SystemConfig`) en base,
- projeter les données dans un `StatusDto` (Domain),
- fournir au reste de l'application l'état global :
    - `Initializing` (true/false),
    - `LastSyncUtc`.

🔹 SystemConfigRepository

Implémente `ISystemConfigRepository`.

Rôle :

- encapsuler la lecture/écriture de `SystemConfig`,
- garantir qu'il n'existe qu'un seul enregistrement de configuration,
- offrir une API claire au reste de l'Infra.

🔹 UpsertService / PokemonUpsertService

Implémentent les interfaces liées à l'upsert (`IUpsertService`, `IPokemonUpsertService`).

Responsabilités :

- recevoir un batch de `PokemonUpsertItem` (Domain),
- charger les données existantes en base pour éviter les doublons,
- créer/mettre à jour les entités Domain (`Pokemon`, `PokemonStat`, `PokemonFlavor`, `PokemonType`, etc.),
- gérer les insertions en cascade,
- produire un `UpsertBatchResult` détaillant :
    - le nombre de créations,
    - le nombre de mises à jour,
    - les éventuels échecs par item.

Pseudo-code illustratif :

```
public async Task<UpsertBatchResult> UpsertPokemonsAsync(
    IReadOnlyList<PokemonUpsertItem> batch,
    CancellationToken ct)
{
    var result = new UpsertBatchResult();

    foreach (var item in batch)
    {
        try
        {
            await _upsertService.UpsertSingleAsync(item, ct);
            result.AddSuccess(item.PokedexId);
        }
        catch (Exception ex)
        {
            result.AddFailure(item.PokedexId, ex.Message);
        }
    }

    await _db.SaveChangesAsync(ct);
    return result;
}
```

🔹 IdempotencyService

Rôle :

- gérer les clés d'idempotence (`IdempotencyKey`),
- empêcher le retraitement d'un même batch ou message,
- tracer les opérations déjà appliquées.

Cela évite :

- les doublons en base,
- les replays non désirés.

4.5 Idempotence et cohérence

L'idempotence est un aspect important de la solution.
Plutôt que de faire confiance à la seule source externe, l'Infrastructure :

- stocke une **clé d'idempotence** (par exemple un identifiant de batch ou de message),
- vérifie si cette clé a déjà été traitée,

- décide de :
    - refuser le retraitement, ou
    - le marquer comme déjà appliqué sans impacter les données.

Ce mécanisme est encapsulé dans :

- l'entité `IdempotencyKey`,
- le service `IdempotencyService`.

L'API n'a pas besoin de connaître le détail : elle s'appuie simplement sur les services d'Infrastructure.

4.6 Résumé du rôle d'Infrastructure

En résumé, `PokeSync.Infrastructure` :

- implémente les contrats définis dans le Domain,
- encapsule l'accès à la base de données,
- gère l'upsert en masse et ses résultats,
- assure l'idempotence et la cohérence,
- fournit des services simples à consommer pour l'API.

C'est la couche qui transforme les **intentions métier** en **actions concrètes**.

API

5. PokeSync.Api

La couche **PokeSync.Api** est la façade HTTP et le point d'entrée opérationnel de PokeSync.
Elle ne se limite pas aux contrôleurs : elle inclut également des **middlewares**, des **services hébergés** (hosted services) et de la **validation** avancée pour sécuriser et fiabiliser les flux.

Son rôle est de :

- exposer les endpoints HTTP (publics et internes),
- valider les payloads entrants (FluentValidation),
- appliquer des préoccupations transverses (correlation id, token interne, idempotence),
- déclencher certains traitements au démarrage ou de manière planifiée (hosted services),
- déléguer toute logique métier à Domain + Infrastructure.

5.1 Structure du projet API

```
PokeSync.Api
├── Contracts
│   ├── GenerationDto.cs
│   ├── TypeDto.cs
│   ├── Idempotency
│   │   └── IdempotencyStatusResponse.cs
│   └── Upsert
│       ├── PokemonUpsertItemDto.cs
│       └── Validation
│           └── PokemonUpsertItemDtoValidator.cs
│
├── Controllers
│   ├── StatusController.cs
│   ├── IdempotencyController.cs
│   └── InternalUpsertController.cs
│
├── HostedServices
│   ├── BootstrapService.cs
│   ├── ConfigurationValidatorHostedService.cs
│   └── NightlySyncService.cs
│
├── Middlewares
│   ├── CorrelationIdMiddleware.cs
│   ├── InternalTokenMiddleware.cs
│   └── IdempotencyMiddleware.cs
│
├── Options
│   └── SecurityOptions.cs
│
├── logs
│   └── pokesync-log-*.json (exemples de journaux)
│
├── Program.cs
├── appsettings*.json
└── Properties/launchSettings.json
```

5.2 Contrôleurs & endpoints

🔹 5.2.1 `GET /api/status`

Contrôleur : `StatusController`
Accès : public (`[AllowAnonymous]`)

Objectif : retourner un statut synthétique du système, basé sur `SystemConfig` :

- `Initializing` : le système est-il encore en phase d'initialisation ?
- `LastSyncUtc` : date de la dernière synchronisation réussie.

Le contrôleur appelle `IStatusService` (Infrastructure), qui lit la base et renvoie un `StatusDto` (Domain), mappé ensuite vers un `StatusResponse` (API).

🔹 5.2.2 `GET /internal/idempotency/status`

Contrôleur : `IdempotencyController`
Route : `internal/idempotency/status`

Objectif : exposer un état lié aux clés d'idempotence et à leur utilisation.
Le contrôleur s'appuie sur `PokeSyncDbContext` pour lire les entrées d'idempotence et projeter le résultat dans un `IdempotencyStatusResponse`.

Ce endpoint est pensé pour un usage **interne** : supervision, debug, orchestrateurs.

🔹 5.2.3 `POST /internal/upsert/types`

Contrôleur : `InternalUpsertController`
Route : `internal/upsert/types`

Objectif : recevoir un batch de `TypeDto` et effectuer un upsert des types élémentaires (`ElementType`).

Le contrôleur :

- normalise les noms (trim + lower-case),
- appelle `IUpsertService.UpsertAsync<ElementType, TypeDto, string>(...)`,
- renvoie un JSON indiquant le nombre d'insertions et le nombre d'éléments ignorés.

🔹 5.2.4 `POST /internal/upsert/generations`

Contrôleur : `InternalUpsertController`
Route : `internal/upsert/generations`

Objectif : synchroniser les générations (`Generation`).

Le contrôleur :

- filtre les entrées invalides (numéro > 0, nom non vide),
- normalise les noms,
- délègue l'upsert à `IUpsertService`,
- retourne aussi un résumé `inserted/skipped`.

🔹 5.2.5 `POST /internal/upsert/pokemons`

Contrôleur : `InternalUpsertController`
Route : `internal/upsert/pokemons`

Objectif : réceptionner un **batch de Pokémon** (`List<PokemonUpsertItemDto>`) et déclencher un upsert complet.

Le contrôleur :

- mappe `PokemonUpsertItemDto` → `PokemonUpsertItem` (model Domain),
- délègue le traitement à `IPokemonUpsertService`,
- obtient un `UpsertBatchResult` détaillant réussite/échec par élément,
- renvoie :
    - `200 OK` si tout est OK,
    - `207 Multi-Status` si certains éléments ont échoué.

Ce route est également la cible principale de la logique d'**idempotence** (cf. middleware dédié plus bas).

5.3 DTO & Validation (FluentValidation)

Les DTO de l'API sont regroupés dans le dossier `Contracts`.
Ils décrivent la forme des données attendues par les endpoints, indépendamment des entités persistées.

Exemple simplifié :

```
public sealed class PokemonUpsertItemDto
{
    public int ExternalId { get; set; }
    public int Number { get; set; }
    public string Name { get; set; } = string.Empty;
    public int GenerationNumber { get; set; }
    public List<string> Types { get; set; } = new();
    public decimal Height { get; set; }
    public decimal Weight { get; set; }
    public string? SpriteUrl { get; set; }
    public List<PokemonStatItemDto> Stats { get; set; } = new();
    public List<PokemonFlavorItemDto> Flavors { get; set; } = new();
}
```

La validation métier « côté API » est assurée par **FluentValidation** via `PokemonUpsertItemDtoValidator`.

Exemples de règles :

- `ExternalId` doit être strictement positif,
- `Number` doit être strictement positif,
- `Name` est obligatoire et a une longueur maximale,
- les noms de stats doivent appartenir à une liste autorisée (`hp`, `attack`, `defense`, `special-attack`, `special-defense`, `speed`),
- etc.

Cette validation permet de :

- protéger l'Infra et le Domain de données mal formées,
- donner des erreurs explicites au producteur de données.

Exemple de gestion des erreurs :

Prenons un lot de 100 Pokémon :

- Cas A : **1 Pokémon** dans le lot n'a pas son champ `Name` renseigné (échec **FluentValidation**). Géré par **FluentValidation** dans `PokemonUpsertItemDtoValidator`. Statut renvoyé : `400 Bad Request`
- Cas B : **1 Pokémon** dans le lot échoue à l'**UpsertService** (ex : erreur d'écriture SQL sur cet item). Le DTO passe la validation (FluentValidation OK). L'`UpsertBatchResult` contient : des succès et au moins un échec (→ `FailedCount > 0`). Statut renvoyé : `207 Multi-Status`.

5.4 Middlewares

Les middlewares de PokeSync.Api gèrent des préoccupations **transverses**.

🔹 5.4.1 `CorrelationIdMiddleware`

- Ajoute un **X-Correlation-Id** à chaque requête (si absent),
- le propage dans le contexte,
- permet de corréler les logs côté API et côté intégrations.

🔹 5.4.2 `InternalTokenMiddleware`

- Vérifie la présence d'un header `X-Internal-Token`,
- compare la valeur reçue à un token attendu (`Security:InternalToken` dans la config),
- retourne `401` ou `403` en cas de token absent ou incorrect.

Ce middleware permet de **sécuriser les endpoints internes** (`/internal/...`) sans mettre en place un système d'auth complet.

🔹 5.4.3 `IdempotencyMiddleware`

- Cible les requêtes `POST/PUT/PATCH` sur le chemin `/internal/upsert/pokemons`,
- lit le corps de la requête (dans une limite définie, ex : 5 Mo),
- calcule une clé d'idempotence (ex : hash du corps ou identifiant fourni),
- interagit avec `IIdempotencyService` pour :
    - rejeter un doublon déjà traité, renvoie d'un statut code 200
    - ou enregistrer une nouvelle requête.
    - en cas de conflit (payload différent de celui enregistré pour un même idempotency key, renvoie un statut 409 qui sera traité côté Mulesoft avec message JSON explicite.)

Ce middleware assure que les appels d'upsert Pokémon restent **idempotents** malgré d'éventuels replays côté producteur.

5.5 Hosted Services

Les hosted services exécutent des tâches **en arrière-plan**, en dehors du cycle de vie direct d'une requête HTTP.

🔹 5.5.1 `BootstrapService`

- S'exécute au démarrage de l'application.
- Vérifie si les tables de référence (Types, Generations) sont déjà peuplées.
- Utilise `IStatusService` pour gérer un **verrou logique** (éviter plusieurs bootstraps concurrents).
- Selon la configuration :
  - en **prod** : prévoit un appel MuleSoft (`POST /exp/pokemons/init`),
  - en **dev/local** : utilise un fichier `seed-data.json` comme source de données.
- Met à jour l'état global (succès / échec) dans `SystemConfig`.

BootstrapService lui-même ne lit pas directement un champ de `SystemConfig` pour ça :

il regarde **`hasTypes`** **et** **`hasGenerations`** dans la base.

```
var hasTypes = await db.Types.AsNoTracking().AnyAsync(stoppingToken);
var hasGenerations = await db.Generation.AsNoTracking().AnyAsync(stoppingToken);

if (hasTypes && hasGenerations)
{
    _logger.LogInformation("Bootstr                                  nerations already populated.");
    return;
}
```


La logique de "ne pas relancer un bootstrap qui est déjà en cours ou déjà réussi" est portée par :

- `SystemConfig.State` (valeur `"ready"`),
- **et** `SystemConfig.BootstrapInProgress` (true/false), via `StatusService.TryBeginBootstrapAsync`.

🔹 5.5.2 `ConfigurationValidatorHostedService`

- Lit la configuration au démarrage (connection string, token interne, origins CORS, etc.).

- Valide les valeurs critiques.
- Ne bloque pas le démarrage, mais **log des warnings/erreurs** en cas de configuration incohérente.

🔹 5.5.3 `NightlySyncService`

- Service planifié (squelette) censé exécuter une synchronisation nocturne.
- Utilise `IStatusService` pour éviter les chevauchements.
- À ce stade, le code est principalement un **stub** : la structure de planification est en place, l'appel réel d'intégration MuleSoft restant à compléter.

5.6 Options & configuration

`SecurityOptions` regroupe les paramètres liés à la sécurité interne :

- token partagé avec MuleSoft (`InternalToken`),
- éventuelles origines autorisées (CORS front),
- autres paramètres de sécurité internes.

Cette classe est bindée sur la section `"Security"` de `appsettings*.json`, via le mécanisme d'options de .NET.

Les fichiers `appsettings.json` et variantes (`Development`, `Production`) définissent :

- la chaîne de connexion SQL Server,
- le token interne,
- le mode de bootstrap (`Bootstrap:Mode` = MuleSoft / LocalSeed),
- d'autres paramètres techniques.

5.7 Résumé du rôle de l'API

PokeSync.Api n'est pas qu'un simple ensemble de contrôleurs :

- elle **valide** les données entrantes,
- elle **sécurise** l'accès via token interne,
- elle **corrèle** les requêtes pour le suivi et le debug,
- elle assure l'**idempotence** sur les opérations critiques,
- elle **prépare** et **valide** l'environnement au démarrage,
- elle offre des points de supervision internes.

Toutefois, elle reste fidèle à la philosophie de la solution :
👉 **aucune logique métier lourde ne vit dans l'API**.
Elle délègue le travail à Domain + Infrastructure, et se concentre sur l'exposition HTTP et les préoccupations transverses.

Vue d’ensemble & Scénarios d’usage

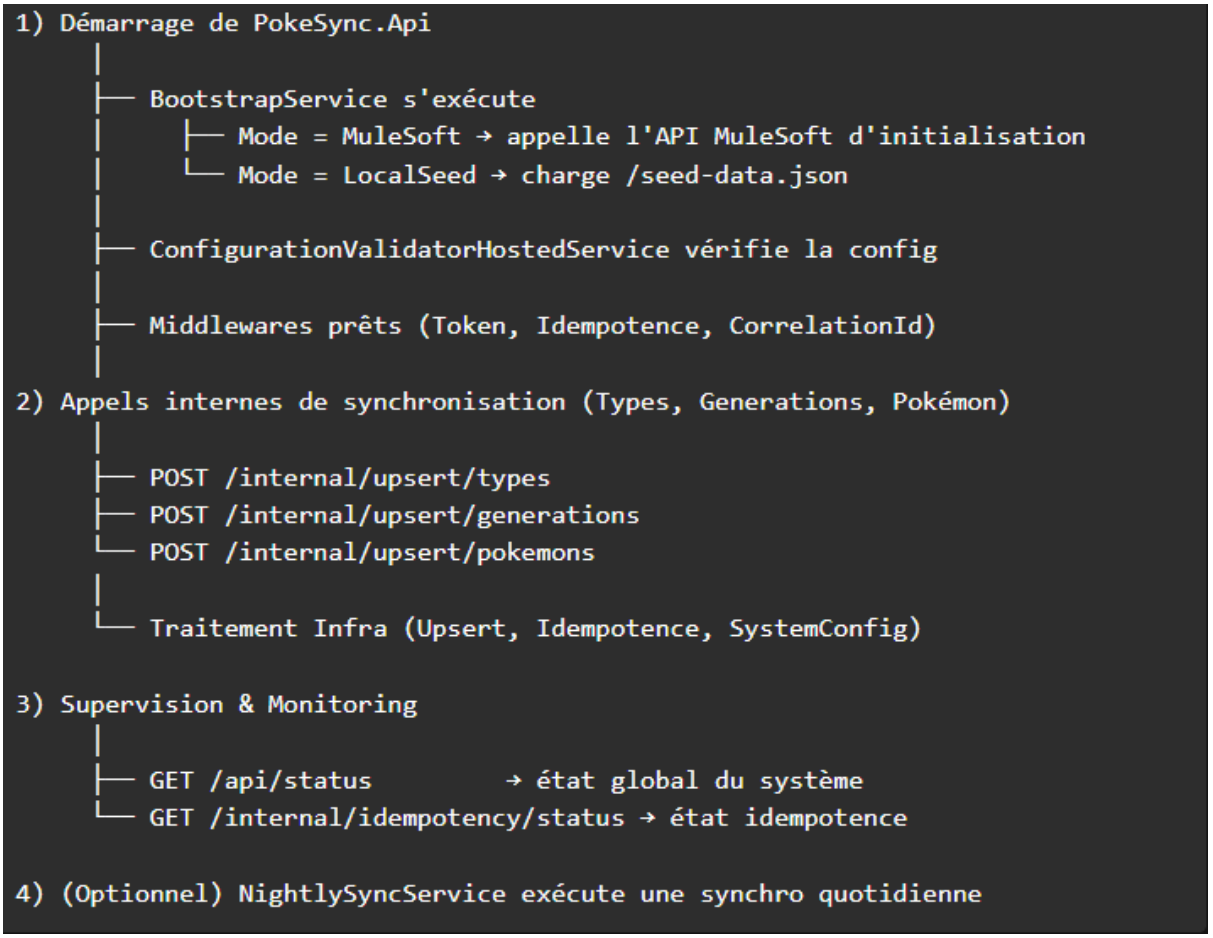
6. Vue d’ensemble & Scénarios d’usage

Cette section synthétise **comment PokeSync s’utilise réellement**, du démarrage à la synchronisation quotidienne, en couvrant la perspective d’un intégrateur (ex : MuleSoft) ou d’un développeur qui exploite l’API.

L’objectif est de fournir une vision simple mais complète des **flux**, **étapes attendues**, et du **fonctionnement global**.

6.1 Cycle de vie complet : démarrage → synchronisation → supervision

Voici la chronologie typique d’un cycle PokeSync :



Ce schéma permet de comprendre le rôle de chaque couche dans le temps.

NB : L’optionnel sera développé en fin de MVP.

6.2 Utilisation côté intégrateur (ex : MuleSoft)

Un intégrateur externe doit respecter **trois règles fondamentales** :

🔸 Règle 1 : Utiliser le token interne

Toute requête vers les endpoints `/internal/*` doit inclure :

```
X-Internal-Token: <token défini dans appsettings>
```

Sans cela, `InternalTokenMiddleware` rejette la requête.
A propos du Token :

en dev → `dotnet`	`user-secrets`
en prod → variables	d'environnement / gestionnaire de secrets

dotnet user-secrets init

dotnet user-secrets set "Security:InternalToken" "super-cle-interne-locale-pour-mulesoft"

Powershell : pour la token
[Convert]::ToBase64String((1..48 | ForEach-Object { Get-Random -Minimum 0 -Maximum 256 }))

OU

-join ((48..57) + (65..90) + (97..122) | Get-Random -Count 64 | ForEach-Object {[char]$_})

🔸 Règle 2 : Envoyer les données dans le bon ordre

Les synchronisations doivent être appelées dans cet ordre :

1. **Types** → `/internal/upsert/types`
2. **Générations** → `/internal/upsert/generations`
3. **Pokémon** → `/internal/upsert/pokemons`

L'ordre garantit que :

- les types existent avant de créer les Pokémon qui les référencent,
- les générations sont disponibles avant d'être attribuées.

Règle 3 : Envoyer un batch complet de Pokémon

Le endpoint :

```
POST /internal/upsert/pokemons
```

attend une **liste complète** de Pokémon, chacun décrit avec :

- id externe & numéro Pokédex,
- nom,
- génération,
- types,
- stats,
- descriptions.

Les DTO doivent respecter les règles FluentValidation → sinon :

- réponse **400 Bad Request** avec liste d'erreurs.

6.3 Idempotence : comment le producteur doit s'en servir

Le middleware d'idempotence garantit que **la même requête ne sera jamais traitée deux fois**.

Deux stratégies possibles côté intégrateur :

Envoyer un `X-Idempotency-Key` en header

Le producteur peut fournir lui-même une clé d'idempotence :

```
X-Idempotency-Key: sync-2025-02-14-08h00
```

6.4 Suivi & supervision

🔹 `GET /api/status`

Permet de savoir en un clin d'œil :

- si le système est initialisé,
- à quelle date/heure la dernière synchronisation a réussi.

🔹 `GET /internal/idempotency/status`

Expose :

- la liste des clés d'idempotence connues,
- les dates de traitement,
- les batchs récents,
- d'éventuelles anomalies.

🔹 Logs structurés JSON

Les logs générés dans *logs/*.json permettent :

- une corrélation via `X-Correlation-Id`,
- une lecture par un SIEM ou un outil d'analyse,
- un suivi clair des erreurs d'Infra ou de validation.

6.5 Développement & extension de PokeSync

Le projet est conçu pour être facilement extensible :

🔸 Ajouter de nouveaux endpoints

Créer un nouveau controller :

```
[ApiController]
[Route("api/new")]
public class NewController : ControllerBase
```

🔸 Ajouter de nouveaux HostedServices

Créer une classe dérivée de `BackgroundService` et l'enregistrer dans `Program.cs`.

🔸 Ajouter un nouveau middleware

Créer une classe `Invoke(HttpContext)` et l'enregistrer dans le pipeline.

🔸 Ajouter un nouvel upsert

Ajouter :

- un DTO API,
- un model Domain,
- une implémentation `IUpsertService` si besoin,
- une route interne.

6.6 Résumé global

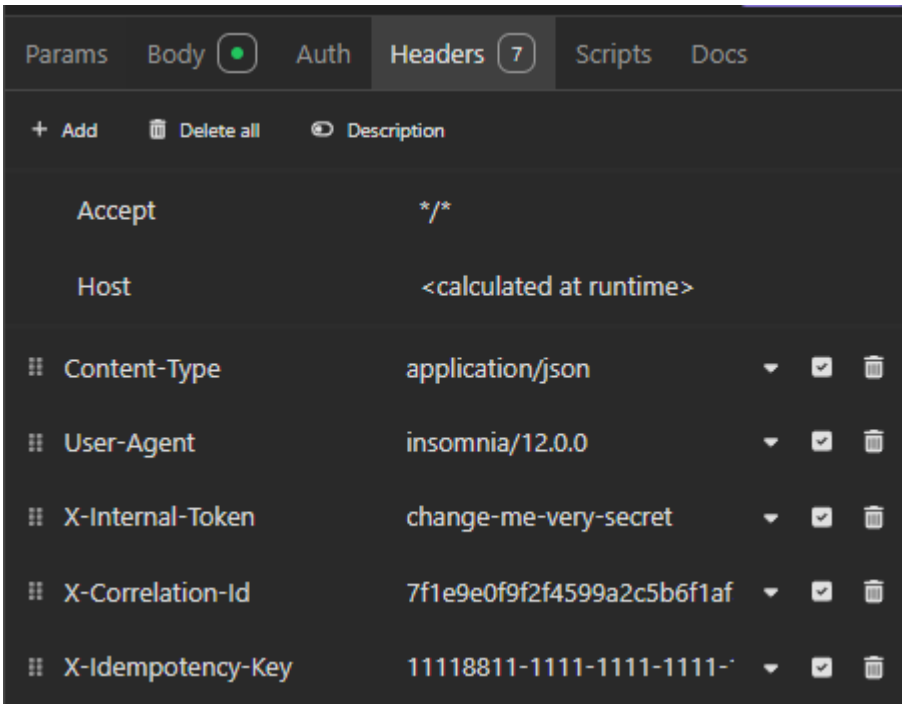
PokeSync offre une architecture claire, robuste et extensible :

- Le **Domain** définit le métier, les interfaces et les modèles.
- L'**Infrastructure** met en œuvre la logique, les upserts, l'idempotence et l'accès SQL.
- L'**API** expose les routes, garantit la validation, la sécurité, la corrélation et l'orchestration.
- Les **HostedServices** automatisent les tâches de démarrage et de synchronisation.

Cette vue d'ensemble donne une compréhension immédiate du fonctionnement de l'ensemble du système.

6.7 Exemple de payload pour test des Endpoints

Header :



POST : http://localhost:<port>/internal/upsert/generations

[{"number": 1 ,"name":"Generation I"},{"number": 2, "name":"Generation II"}]

POST : http://localhost:<port>/internal/upsert/types

[{"name":"water"},{"name": "fire"}]

POST : http://localhost:<port>/internal/upsert/pokemons


```
[
  {
    "externalId": 1,
    "number": 4,
    "name": "ivysaur",
    "generationNumber": 1,
    "spriteUrl":
"https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/2.png",
    "height": 1.0,
    "weight": 13.0,
    "types": ["grass", "poison"],
    "stats": [
      { "name": "hp", "value": 60 },
      { "name": "attack", "value": 62 },
      { "name": "defense", "value": 63 },
      { "name": "special-attack", "value": 80 },
      { "name": "special-defense", "value": 80 },
      { "name": "speed", "value": 60 }
    ],
    "flavors": [
      { "language": "en", "text": "When the bulb on its back grows large, it appears to lose the
ability to stand on its hind legs." }
    ]
  }
]
```