PokéSync – Integration Showcase (MuleSoft × .NET × Angular)

Phase 0 — Cadrage & Besoin (version unifiée)

0.1 Pourquoi ce projet ? (Vision)

- **But pédagogique** : démontrer un flux d'intégration LED complet (MuleSoft ↔ .NET ↔ SQL ↔ Angular) sur un thème ludique et structuré.
- **But professionnel** : présenter une architecture pensée comme en production (idempotence, reprise, observabilité, sécurité).
- **Livrable attendu** : démo fonctionnelle + README + schéma d'architecture + OpenAPI + (option) export Postman.

0.2 Besoin fonctionnel

- **Consulter** des Pokémon par **génération** et **type**.
- **Lister** les Pokémon (pagination) et **afficher les détails** : stats, sprite, types, et **descriptions Pokédex selon la génération**.
- **Afficher les images officielles** et (optionnellement) **les sprites générationnels** issus des différentes versions du jeu.
- **Fiabilité** : initialisation par batch (par génération), synchronisation delta nocturne.

0.3 Indicateurs de réussite (KPIs)

- **Intégration initiale** : la génération 1 doit être importée en moins de **5 minutes**, avec **moins de 1 % d'erreurs**, toutes rejouées automatiquement.
- **Performance API** : le temps de réponse moyen (`TTFB`) pour l'appel `/api/pokemons` doit rester **inférieur à 200 ms** sur l'environnement local/NAS.
- **Synchronisation nocturne** : la mise à jour delta planifiée (NightlySync) doit s'exécuter en **moins de 10 minutes**, incluant la maintenance des index SQL.

(Mesure via logs MuleSoft + .NET, endpoint `/api/status`*, ou widget Angular de supervision.)*

0.4 Périmètre / Hors périmètre

- **In** : Types, Générations, Pokémon (**stats**, **sprite officiel**, **types**, **descriptions Pokédex par génération**).
- **Out** : Movesets détaillés, localisations, rencontres, capacités étendues, images multiples (formes régionales, shiny, etc.).

0.5 Contraintes & Risques

- **Quotas et limitations API** : la PokeAPI applique un **rate limit**. Trop de requêtes simultanées renvoient des erreurs `HTTP 429 (Too Many Requests)`. →
  Mesure : implémenter un **throttling** et un **retry avec backoff** côté MuleSoft pour respecter ces quotas.

- **Latence réseau et durée du seed initial** : le premier import (toutes les générations) peut durer plusieurs minutes selon la bande passante et la charge serveur. →
  Mesure : traitement **par génération** et **par batchs** (100–200 éléments) avec possibilité de reprise en cas d'interruption.

- **Idempotence et erreurs partielles** : en cas de coupure ou d'erreur lors d'un batch, on risque des doublons ou des données incomplètes. →
  Mesure : utilisation d'un identifiant unique `ExternalId` + envoi de `X-Idempotency-Key` pour éviter tout doublon.

- **Évolution du schéma PokeAPI (mapping)** : risque que le format JSON de PokeAPI change (renommage, déplacement ou suppression de champs). →
  Mesure : définir un **mapping DTO versionné** côté MuleSoft et prévoir un **test d'intégration de validation du schéma** pour détecter rapidement les ruptures.

- **Disponibilité externe** : dépendance forte à une API publique sans SLA garanti. →
  Mesure : possibilité future d'utiliser un **cache local** ou un **miroir PokeAPI** en cas d'indisponibilité prolongée.

0.6 Solution technique choisie

A) Backend .NET (source of truth)

⚙️Endpoints publics (ouverts au front Angular)

Endpoint	Méthode	Rôle	Détails
`GET /api/status`	GET	Vérifie si le système est prêt.	Renvoie `{ initializing: boolean, lastSyncUtc: datetime }`. Si `initializing = true`, le front affiche l'écran de maintenance.
`GET /api/types`	GET	Liste les types Pokémon.	Sert à remplir le filtre "Type" du front. Les données proviennent de la table `Type`.
`GET /api/generations`	GET	Liste les générations disponibles.	Sert au filtre "Génération" du front. Ex. Generation 1 à 9.
`GET /api/pokemons`	GET	Liste paginée des Pokémon.	Paramètres `typeId`, `generationId`, `page`, `pageSize`. Permet de filtrer, trier et paginer les résultats.

Endpoint	Méthode	Rôle	Détails
`GET /api/pokemons/{id}`	GET	Détail complet d'un Pokémon.	Récupère le Pokémon, ses stats, types et descriptions Pokédex (si disponibles).

🔒 **Endpoints internes (réservés à MuleSoft)**

Ces routes ne sont pas publiques : elles servent exclusivement à alimenter la base de données à partir des données traitées par MuleSoft. Elles nécessitent un **header sécurisé** `X-Internal-Token` pour être exécutées.

Endpoint	Méthode	Rôle	Détails
`POST /internal/upsert/types`	POST	Met à jour les types Pokémon.	Reçoit un tableau `[{ name }]`. Si le type existe déjà, il n'est pas recréé.
`POST /internal/upsert/generations`	POST	Met à jour les générations.	Reçoit un tableau `[{ number, name }]`. Idempotent : ne duplique pas les entrées existantes.
`POST /internal/upsert/pokemons`	POST	Crée ou met à jour les Pokémon.	Reçoit un batch JSON (100–200 éléments) avec les Pokémon normalisés. Le backend gère l'upsert, les clés étrangères, les stats et les types associés.

🧰**Résumé du comportement**

- Les **endpoints publics** servent à **afficher** les données au front Angular.
- Les **endpoints internes** servent à **alimenter** la base de données via MuleSoft.
- Authentification : CORS restreint pour le front, `X-Internal-Token` ou mTLS pour MuleSoft.
- Tous les endpoints d'écriture sont **idempotents** pour permettre la reprise sans doublons.

B) MuleSoft (Architecture LED)

🧱**1. Experience API (EXP)**

Interface d'entrée de l'écosystème MuleSoft, exposée au backend .NET ou à d'autres services externes. Elle ne contient **aucune logique métier** — elle reçoit la demande et la transmet à la Process API.

Endpoint	Méthode	Description
`POST /exp/pokemons/init`	Lance la synchronisation initiale.	Déclenche la récupération complète (par génération) des Pokémon et l'insertion en base.
`POST /exp/pokemons/delta`	Lance la synchronisation nocturne (delta).	Compare les données distantes et locales, puis met à jour les différences.

🔒 Authentification : token interne (API Manager ou Header statique). L'EXP renvoie immédiatement un `202 Accepted` et délègue le traitement asynchrone à la PROC.

⚙️2. Process API (PROC)

C'est le **cœur métier** de MuleSoft. Elle orchestre le flux de données entre PokeAPI et le backend .NET.

Rôle	Détails
Orchestration par génération	Boucle sur chaque génération (`1..n`) → appelle la System Pokémon API.
Découpage en lots	Découpe la collecte en **batchs de 100–200 Pokémon** pour éviter surcharge et quotas.
Retry/Backoff	Implémente un retry exponentiel en cas d'erreurs réseau ou `HTTP 429`.
Checkpointing	Stocke la progression `(generation, offset)` dans un **Object Store MuleSoft** pour permettre une reprise après crash.
Transformation (DataWeave)	Mappe les payloads PokeAPI → DTO normalisés (.NET).
Appel System .NET API	Envoie les batchs au backend via `POST /internal/upsert/*`.
Logging & Monitoring	Utilise un **Correlation-Id** pour tracer chaque lot dans Mule + .NET.

🛢️3. System Pokémon API

Couche « source de vérité externe ». Elle encapsule tous les appels à la **PokeAPI publique**, gère la pagination et les erreurs.

Endpoint consommé	Rôle	Détails
`GET /type`	Référentiel des types.	Récupéré une fois au démarrage du flux d'init.
`GET /generation/{n}`	Liste des Pokémon d'une génération.	Fournit les URLs vers les `pokemon_species`.
`GET /pokemon/{id}`	Données détaillées.	Fournit types, stats, sprites, taille/poids.
`GET /pokemon-species/{id}` *(fallback)*	Fallback pour trouver la variety par défaut.	Utilisé quand `/pokemon/{id}` échoue.

> **Optimisation** : les appels `/pokemon` sont parallélisés (max 8–12 simultanés), avec gestion des `429`.

🔒4. System .NET API

Dernière couche d'intégration — elle assure la communication sécurisée avec le backend.

Endpoint cible	Sécurité	Rôle
`POST /internal/upsert/types`	Header `X-Internal-Token` ou mTLS	Envoi du référentiel de types.
`POST /internal/upsert/generations`	idem	Envoi du référentiel des générations.
`POST /internal/upsert/pokemons`	idem	Envoi des batchs Pokémon normalisés (100–200).

Tous les appels sont **idempotents** : MuleSoft peut rejouer un lot sans créer de doublons.

🧰**Résumé global**

Couche	Rôle principal	Interaction
EXP	Interface d'entrée (trigger d'intégration)	.NET ↔ MuleSoft
PROC	Orchestration métier + mapping + retry	EXP ↔ SYS
SYS_Pokemon	Source externe (PokeAPI)	Mule ↔ PokeAPI
SYS_.NET	Cible interne (backend)	Mule ↔ ASP.NET

C) Données & Modèle

🧰**Structure globale**

Le modèle de données a été conçu pour **garantir la flexibilité, la normalisation et les performances de requête**. Chaque entité correspond à un **concept stable** de l'univers Pokémon, tandis que les **tables de liaison** assurent la gestion des relations *many-to-many* (ex : un Pokémon peut avoir plusieurs types).

Table	Rôle	Détails
`Type`	Référentiel des types Pokémon.	Exemple : Feu, Eau, Plante, Acier, etc. Données statiques récupérées via `GET /type`.
`Generation`	Référentiel des générations.	Contient le nom et le numéro (`generation-i` → 1). Sert de point d'entrée pour la synchronisation.
`Pokemon`	Entité principale.	Contient les métadonnées de base : `externalId`, `number`, `name`, `spriteUrl`, `generationId`, `height`, `weight`, etc.
`PokemonType`	Table de liaison entre `Pokemon` et `Type`.	Permet de gérer les Pokémon à double type (ex. Eau/Glace, Sol/Roche). Indexée sur les deux colonnes pour accélérer les filtres.

Table	Rôle	Détails
`PokemonStat`	Détail des statistiques d'un Pokémon.	Une ligne par couple (`pokemonId`, `statName`). Ex. { "hp", 45 } ou { "speed", 80 }. Permet de générer dynamiquement des tableaux ou graphiques.
`PokemonFlavor` *(optionnelle)*	Texte Pokédex par génération.	Une ligne par couple (`pokemonId`, `generationId`). Permet de présenter les descriptions selon la version du jeu.

🧮 DTO normalisé (côté intégration)

Format standardisé utilisé par MuleSoft et .NET pour insérer ou exposer les Pokémon. Ce format simplifie la désérialisation, le contrôle de version et la génération automatique des entités.

```
{
  "externalId": "25",
  "number": 25,
  "name": "pikachu",
  "generation": 1,
  "types": ["electric"],
  "spriteUrl": "https://raw.githubusercontent.com/.../official-artwork/
25.png",
  "stats": {
    "hp": 35,
    "attack": 55,
    "defense": 40,
    "special-attack": 50,
    "special-defense": 50,
    "speed": 90
  },
  "flavorTexts": [
    { "generation": 1, "text": "..." },
    { "generation": 3, "text": "..." }
  ]
}
```

> Ce DTO correspond à la donnée "normalisée" avant insertion en base et reflète les besoins du front Angular (affichage par génération et filtres dynamiques).

🔍 Requêtes clés

- **Filtrage par type** → jointure `PokemonType` ↔ `Type`.
  `SELECT * FROM Pokemon p JOIN PokemonType pt ON p.Id = pt.PokemonId WHERE pt.TypeId = X`
- **Filtrage par génération** → filtre direct `WHERE p.GenerationId = X`.

- **Pagination & tri** → index sur `(GenerationId, Id)` et `(Name)` pour performances optimales.
- **Préchargement des types & stats** → relations `Include()` dans EF Core pour réduire le nombre de requêtes.

🕓Schéma (à venir)

Le modèle pourra être représenté sous forme de diagramme relationnel (Merise ou crow-foot), avec : - une cardinalité **1-N** entre `Generation` → `Pokemon`,
- une **N-N** entre `Pokemon` ↔ `Type` (via `PokemonType`),
- une **1-N** entre `Pokemon` → `PokemonStat`,
- une **1-N** entre `Pokemon` → `PokemonFlavor`.

D) Front Angular (UI)

🪠Structure & routing

- **Standalone components** (Angular 18), pas de NgModule.
- **Routes** :
- `/` → redirige vers `/pokedex`
- `/pokedex` (liste) avec **query params** : `type`, `generation`, `page` (ex: `/pokedex?type=fire&generation=1&page=3`)
- `/pokemon/:id` (détail direct, deep-link) → ouvre le **pane de détail** sur la grille sélectionnée.

🧰Composants (standalone)

- `AppShellComponent` : header minimal, conteneur de routes.
- `MaintenanceComponent` : affiche l'écran maintenance si `initializing=true` (via `/api/status`).
- `PokedexPageComponent` : page principale (orchestration filtres + grille + détail).
- `TypeFilterComponent` : liste/checkbox des types (badges cliquables).
- `GenerationFilterComponent` : select ou pills 1..N.
- `PokemonGridComponent` : grille responsive (cards), écoute pagination.
- `PokemonCardComponent` : **Number**, **Name**, **Sprite**, **Types** (badges). Clique → ouvre détail.
- `PokemonDetailComponent` : pane/modal latéral (ou route dédiée) avec **stats** + **flavor texts** (par génération).
- `StatsTableComponent` *(option)* : tableau des 6 stats.
- `StatsRadarComponent` *(option)* : radar chart (recharts) des 6 stats.
- `PaginationComponent` : contrôles page précédente/suivante (ou **infinite scroll** en option).
- `LoadingSkeletonComponent` : placeholders pour grille/détail.
- `ErrorBannerComponent` / `EmptyStateComponent` : messages d'erreur et d'absence de résultats.

🗃️Modèles front (DTO UI)

- `PokemonListItem` : `{ id, number, name, types: string[], spriteUrl }`

- `PokemonDetail` : `{ ...ListItem, generation, stats: {hp,atk,def,spa,spd,spe}, flavorTexts: {generation: number, text: string}[] }`

🔌**Services & interceptors**

- `ApiService` : appels `/api/status`, `/api/types`, `/api/generations`, `/api/pokemons`, `/api/pokemons/{id}`.
- `StoreService` (RxJS Signals) :
- `types$`, `generations$` (prefetch au boot)
- `pokemonsCache` key = `type|gen|page` avec **TTL** (ex. 10 min)
- `selectedPokemonId` et `detail$` (memoized)
- `CorrelationIdInterceptor` : ajoute `X-Correlation-Id` à chaque requête.
- `HttpErrorInterceptor` : mappe erreurs (réseau/serveur) → `ErrorBanner`.

UX & comportements

- Au boot : appelle `/api/status` → si initializing → `MaintenanceComponent` (front **ne** spamme pas les autres APIs).
- Filtres = **query params** (URL shareable). Changement de filtre → reset `page=1`.
- **Pagination** : 20 items/page (par défaut), tri par `number` asc.
- **Prefetch** : en fin de page `n`, précharge `n+1` (même filtre) dans le cache.
- **Cache invalidation** : TTL expiré ou changement de filtre → refetch.
- **Accessibilité** : navigation clavier (focus visible), ARIA sur filtres, contrastes suffisants.

🧮**Performance**

- Change detection `OnPush` sur tous les composants d'affichage.
- `trackBy` sur les listes.
- Lazy-load des images (attribute `loading="lazy"`).
- **Debounce** (250 ms) sur les changements de filtres pour limiter les requêtes.

🧪**Tests (light)**

- Unit tests : `ApiService` (mocks HTTP), `StoreService` (TTL & cache), rendering `PokemonCard`.
- E2E (option) : scénario "filtre type + gen → pagination → détail → back".

E) Sécurité & Ops

1) Sécurité des APIs

- **Front ↔ .NET (public)**
- **CORS** strict (origines autorisées : domaine front uniquement).
- **TLS obligatoire** (https partout).
- **Rate limiting** basique côté .NET (ex. 60 req/min par IP sur `/api/*`).
- **MuleSoft ↔ .NET (interne)**
- **Header** `X-Internal-Token` signé, rotation possible (clé active + clé précédente).
- **Option mTLS** (certificat client Mule validé côté .NET).
- **IP allowlist** (option) si déploiement en LAN/VPN/NAS.
- **Idempotence**

- Contrainte `UNIQUE(ExternalId)` sur Pokémon.
- `X-Idempotency-Key` par batch côté Mule ; table `IdempotencyKey` (clé, horodatage, hash du payload, TTL 48h).
- **Secrets management**
- Aucun secret dans le code ou les fichiers du build.
- Utilisation de **variables d'environnement** ou **Docker secrets** pour les connexions, tokens et mots de passe.
- En local : `dotnet user-secrets`.
- En prod/NAS : variables d'environnement ou secrets Docker.
- MuleSoft : **Secure Configuration Properties** ou **Anypoint Secrets Manager**.

---

2) Conventions d'erreurs & résilience

- **Codes** : `200` / `201` / `202` succès ; `400` validation ; `401/403` auth ; `404` not found ; `409` conflit idempotence ; `429` limite ; `5xx` serveur.
- **Batchs internes (`/internal/*`)** : `207 Multi-Status` possible pour signaler succès/échecs partiels.
- **Retry/backoff** : Mule = exponentiel sur `429/5xx`, respect `Retry-After`.
- **Timeouts** : HTTP client .NET (15s), serveur Kestrel, Mule requester timeouts explicites.
- **Validation** : FluentValidation + ProblemDetails cohérents.

---

3) Observabilité (logs, métriques, traces)

- **Correlation** : `X-Correlation-Id` propagé Front → .NET → Mule.
- **Logs** : Serilog JSON (niveau Info/Warning/Error, enrichers : correlation, durée, endpoint, user-agent).
- **Métriques** : durée, taux d'erreur, inserts/updates, taille des batchs, exposées via `/api/status`.
- **Traces distribuées (option)** : OpenTelemetry (Jaeger/Zipkin) ou App Insights.
- **Dashboards** : Grafana/Prometheus ou Anypoint Monitoring, plus un mini widget Angular d'admin.

---

4) Santé & supervision

- `GET /healthz` (liveness) : process OK.
- `GET /api/status` (readiness) : DB OK, Mule reachable?, initializing, lastSyncUtc, KPIs.
- Alerte si durée de seed > seuil, errorRate > 1 %, ou échec NightlySync.

---

5) Opérations planifiées

- **Hosted Services (.NET)** :
- `BootstrapService` : exécute le seed initial (Types/Generations) si DB vide.
- `NightlySyncService` : CRON 03:00 UTC → delta Mule + maintenance SQL.
- **Maintenance SQL** : rebuild index, update statistics, suivi de durée et fragmentation.
- **Migrations EF Core** : migrations automatiques au démarrage, versionnées et planifiées.

---

6) Sauvegarde & reprise

- **Backups** : SQL locales (NAS snapshots), rétention 7 jours.
- **Reprise** : checkpoints Mule `(generation, offset)` + idempotence côté .NET.
- **Runbook** : procédure de reseed partiel (génération X uniquement).

---

7) Environnements & déploiement

- **Envs** : `dev` (local), `test` (NAS/VM), `prod-demo` (option).
- **Config par env** : CORS, strings de connexion, tokens.
- **Déploiement** : Docker Compose (`api`, `sqlserver`, `jaeger` éventuel). GitHub Releases + scripts PowerShell/Bash.

---

8) Sécurité front

- **Headers** : `Content-Security-Policy` (whitelist `api` & sprites), `Referrer-Policy: no-referrer`.
- **Données** : aucun secret dans le front. Cache contrôlé (`ETag`, `Cache-Control`) sur `/api/types` et `/api/generations`.

---

9) Runbooks & bonnes pratiques

- **Rotation du** `X-Internal-Token` (clé active + clé précédente).
- **Relancer Seed Gen N** via `/exp/pokemons/init?gen=N`.
- **Rejouer batch KO** : relecture `207` + même `X-Idempotency-Key` (ou nouvelle si modifié).
- **Reprise post-incident** : restaurer DB + reprise Mule via checkpoints.
