Documentación
Pokemón Bytes

Documentación – Fase V

Índice

1. Introducción y Objetivos

- Automatización mediante Data Seeding.

2. Arquitectura del Cliente Reactivo

- 2.1. Uso de WebClient vs RestTemplate.
- 2.2. Estrategia ETL (Extract, Transform, Load).

3. Implementación del Data Seeding (DataLoader)

- 3.1. Orquestación Reactiva (Project Reactor y Mono.zip).
- 3.2. Manejo de Buffers y Lotes.

4. Trazabilidad del Código

- 4.1. Configuración de Buffer en PokeApiService.
- 4.2. Lógica de Carga Masiva en DataLoader.

5. Diagramas y Evidencia

- 5.1. Diagrama de Secuencia Asíncrono.
- 5.2 Logs del Sistema

1. Introducción y Objetivos

En un RPG de la escala de Pokémon, gestionar manualmente la información de cientos de criaturas y movimientos es inviable y propenso a errores humanos. El objetivo de esta fase ha sido automatizar la población de la base de datos (**Data Seeding**) conectando el backend con la **PokéAPI** (la fuente de datos pública oficial).

El sistema implementa un cliente HTTP reactivo que, al iniciar el servidor, verifica la integridad de los datos locales y, si es necesario, descarga, procesa y almacena la información de la Generación II (251 Pokémon y Movimientos) de manera autónoma.

Fuente oficial API: https://pokeapi.co/docs/v2#games-section

2. Arquitectura del Cliente Reactivo

Para esta tarea no se utilizó el cliente tradicional (RestTemplate, bloqueante), sino **WebClient** (parte del stack **Spring WebFlux**).

2.1. Justificación

- **Asincronía y Paralelismo:** La carga inicial requiere realizar más de **500 peticiones HTTP** (251 Pokémon + 251 Especies + Movimientos). Hacerlo de forma secuencial llevaría varios minutos de arranque. Con WebClient y Flux, las peticiones se lanzan en paralelo de forma no bloqueante.
- **Gestión de Memoria:** Se configuró un ExchangeStrategies personalizado para aumentar el buffer de memoria a **16MB**, necesario para procesar los grandes JSONs de respuesta de la API sin desbordamientos de pila.

2.2. Flujo de Datos (ETL)

El proceso sigue un esquema **ETL (Extract, Transform, Load)**:

1. **Extract:** Descarga de JSONs crudos desde pokeapi.co.
2. **Transform:** Mapeo de campos anidados (JSON) a la estructura plana de la entidad JPA (PokedexMaestra). Fusión de datos de "Detalles" (Stats) y "Especies" (Ratio de captura).
3. **Load:** Persistencia en MySQL mediante saveAll() en lotes para optimizar el rendimiento de la base de datos.

3. Implementación Data Seeding (DataLoader)

La clase DataLoader.java implementa la interfaz CommandLineRunner, lo que garantiza su ejecución automática justo después de que el contexto de Spring Boot se haya levantado.

3.1. Orquestación Reactiva

El código utiliza operadores de Reactor (Flux, Mono.zip, flatMap) para combinar múltiples fuentes de datos.

Lógica Implementada:

- **Verificación de Estado:** if (pokedexRepository.count() < POKEMON_LIMIT) evita descargas innecesarias si los datos ya existen.

- **Fusión de Peticiones (Mono.zip):** Para cada Pokémon, se necesitan dos endpoints distintos: /pokemon/{id} (para stats y tipos) y /pokemon-species/{id} (para el ratio de captura). El sistema lanza ambas peticiones simultáneamente y combina los resultados en una tupla cuando ambas han finalizado.

3.2. Configuración del Cliente (PokeApiService)

El servicio PokeApiService encapsula la configuración de bajo nivel del cliente HTTP.

4. Trazabilidad del Código

A continuación, los fragmentos clave que demuestran la implementación .

4.1. Configuración de WebClient (PokeApiService.java)

```
@Service
public class PokeApiService {
    private final WebClient webClient;

    public PokeApiService(WebClient.Builder webClientBuilder) {
        // Aumento del límite de buffer a 16MB para soportar JSONs grandes
        final int maxBufferSize = 16 * 1024 * 1024;
        final ExchangeStrategies strategies = ExchangeStrategies.builder()
            .codecs(codecs ->
codecs.defaultCodecs().maxInMemorySize(maxBufferSize))
            .build();

        this.webClient = webClientBuilder
            .baseUrl("https://pokeapi.co/api/v2/")
            .exchangeStrategies(strategies)
            .build();
    }

    // Petición Asíncrona que devuelve un Mono
    public Mono<Map<String, Object>> getPokemonDetails(String name) {
        return webClient.get()
                .uri("pokemon/{name}", name)
                .retrieve()
                  .bodyToMono(new ParameterizedTypeReference<Map<String,
Object>>() {});
```

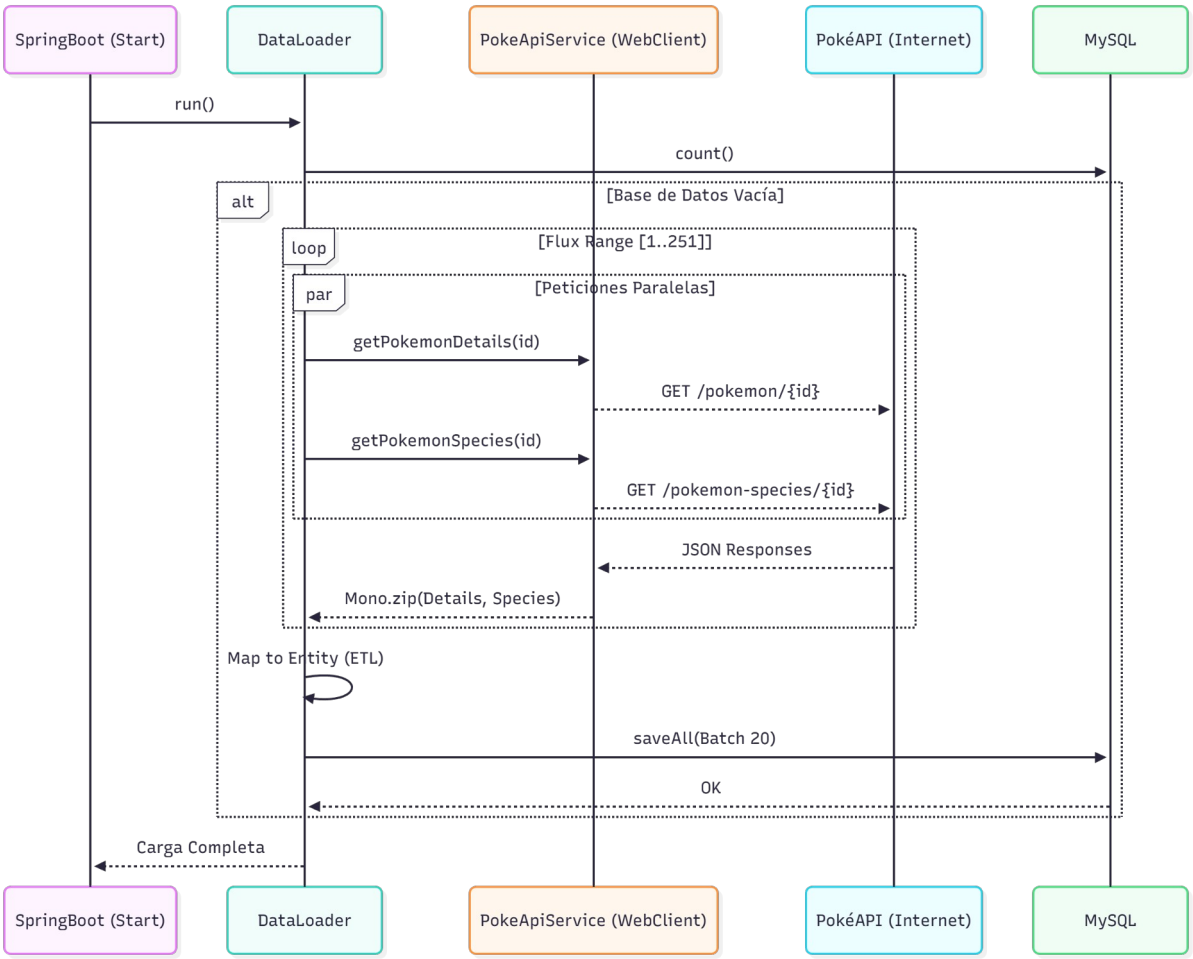
```
    }
}
```

4.2. Lógica de Carga Masiva (DataLoader.java)

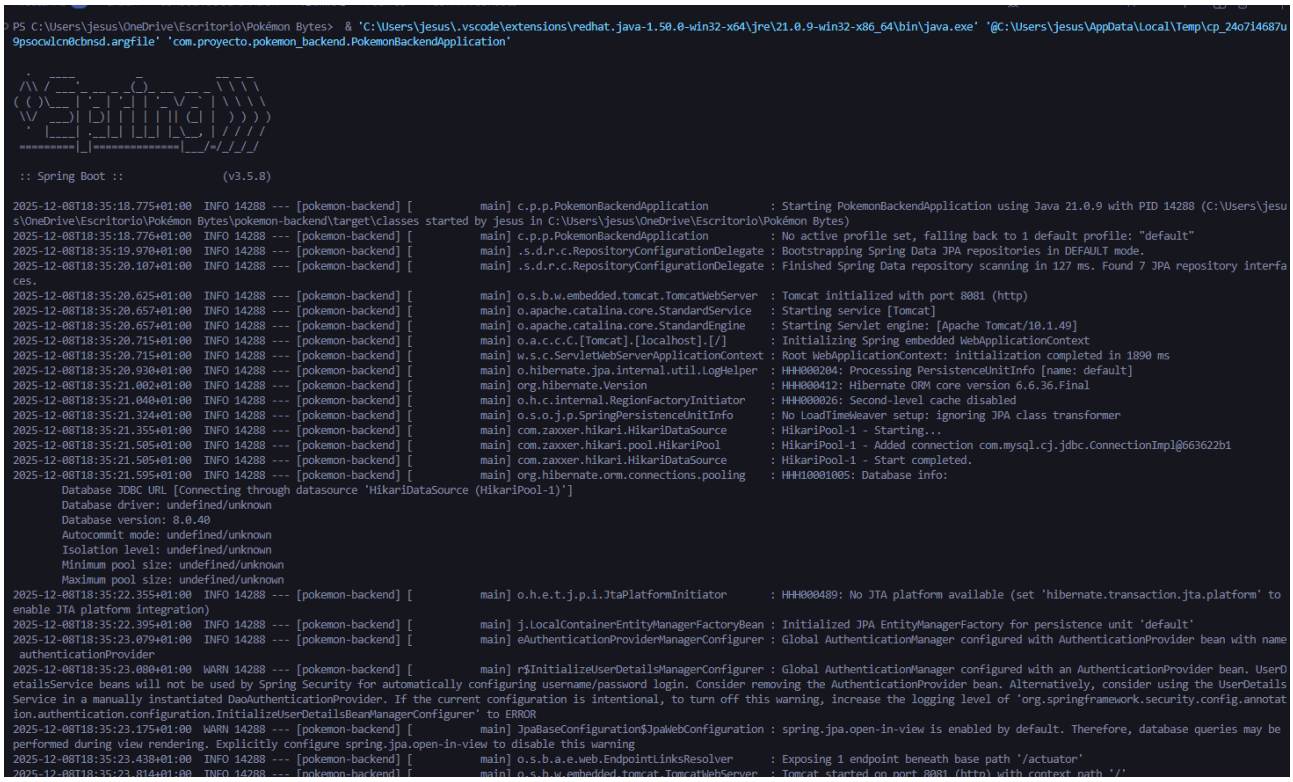
```
@Component
public class DataLoader implements CommandLineRunner {

    // Constantes de Gen II
    private static final int POKEMON_LIMIT = 251;

    @Override
    public void run(String... args) {
        if (pokedexRepository.count() < POKEMON_LIMIT) {

            // Flujo Reactivo: Itera del 1 al 251
            Flux.range(1, POKEMON_LIMIT)
                .flatMap(id ->
                    // Fusión de dos llamadas API en paralelo
                    Mono.zip(
                        apiService.getPokemonDetails(String.valueOf(id)), // T1:
                                                                          Stats
                        apiService.getPokemonSpecies(String.valueOf(id))  // T2:
                                                                          Ratio Captura
                    )
                , 5) // Concurrencia controlada (5 hilos)

                .map(tuple -> mapCombinedDataToEntity(tuple.getT1(),
                 tuple.getT2()))
                .buffer(20) // Agrupa en listas de 20 para inserción
                .doOnNext(pokedexRepository::saveAll) // Persistencia en MySQL
                .blockLast(); // Espera a que termine el proceso antes de liberar
                                                                        el hilo
        }
    }
}
```

5. Diagramas y Evidencia

5.1. Diagrama de Secuencia Asíncrono.



5.2 Logs del Sistema



Captura de la terminal durante el despliegue de PokemonBackendApplication. Se observa inicialización exitosa del Tomcat en el puerto 8081, la conexión a la base de datos MySQL mediante el pool HikariCP y la ejecución de las consultas de verificación de integridad (Hibernate Select Count) de los Seeders.

Análisis de Logs

- *Tomcat initialized with port 8081*: Confirma que la configuración en application.properties (server.port=8081) funciona y no choca con el puerto 8080 estándar.
- *HikariPool-1 - Start completed:* La conexión con MySQL es exitosa. El usuario/contraseña son correctos.
- *Hibernate: select count(*) ...:* Estas líneas finales demuestran que los componentes DataLoader, ItemSeeder y TipoInitializer se están ejecutando.
    - Como solo salen los select count(*), significa que la base de datos ya tiene los datos cargados, por lo que el sistema inteligentemente se salta la descarga de la PokeAPI (evitando perder tiempo).
- *Started PokemonBackendApplication in 5.475 seconds*: El tiempo de arranque es bueno (menos de 6 segundos).