Mutant Detector API

Documentación Técnica y Arquitectura del Sistema

Ignacio Gracia
Desarrollador Backend

23 de noviembre de 2025

Índice

1 Introducción

El presente documento detalla la arquitectura técnica, decisiones de diseño y el flujo de datos del proyecto **Mutant Detector**. Esta API REST ha sido diseñada para identificar secuencias de ADN mutante de manera eficiente, escalable y tolerante a la alta concurrencia, cumpliendo con los requisitos del desafío técnico de MercadoLibre.

2 Arquitectura del Sistema

El proyecto implementa una **Arquitectura en Capas (Layered Architecture)** utilizando el framework Spring Boot 3.2.0 y Java 21. Esta estructura garantiza la separación de responsabilidades y facilita el mantenimiento y testabilidad.

Controller Layer (`org.example.controller`)
: Punto de entrada de la API. Maneja las peticiones HTTP, ejecuta validaciones de entrada (JSR-303) mediante DTOs y gestiona los códigos de respuesta HTTP estandarizados.

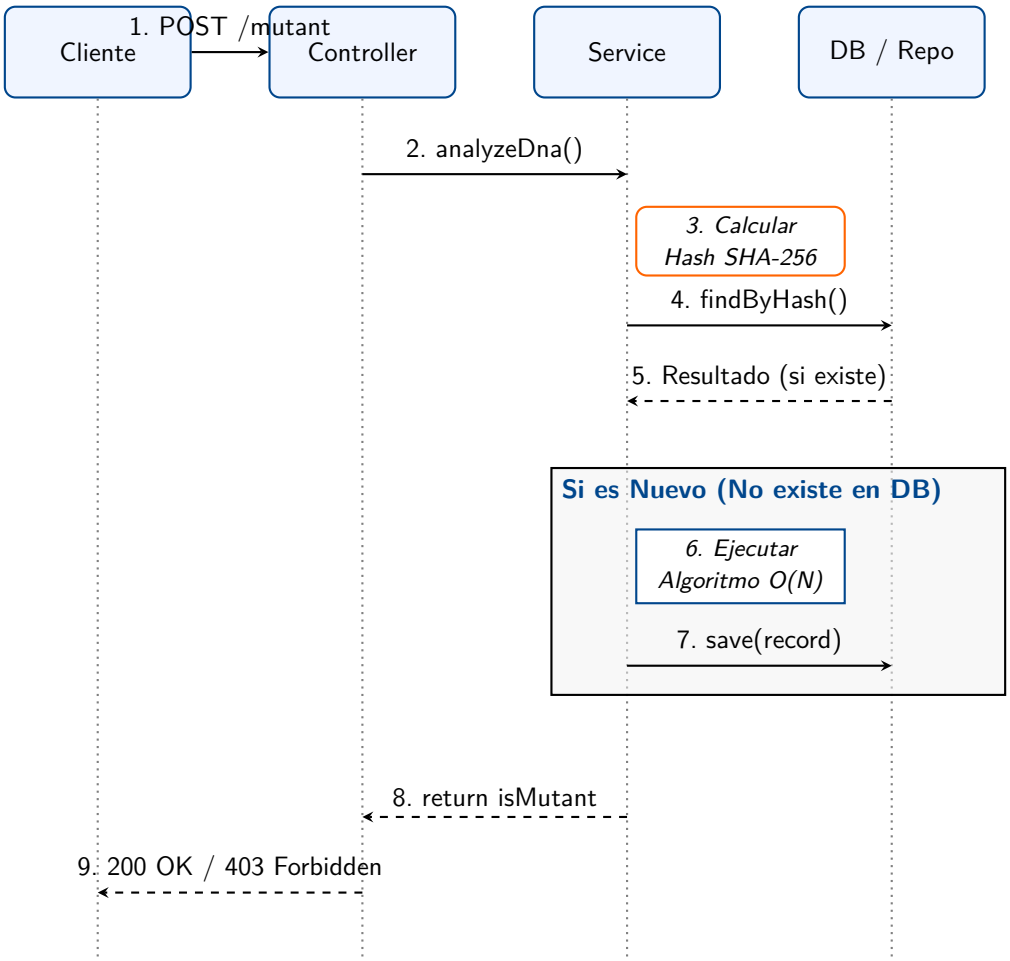
Service Layer (`org.example.service`)
: Contiene la lógica de negocio pura. Aquí reside el algoritmo de detección (`MutantDetector`), la lógica de *hashing* para deduplicación y la orquestación entre el cálculo y la persistencia.

Repository Layer (`org.example.repository`)
: Capa de acceso a datos (DAO). Utiliza Spring Data JPA e Hibernate para interactuar con la base de datos de manera abstracta, permitiendo cambiar el motor de base de datos con mínimo impacto.

Entity Layer (`org.example.entity`)
: Representación del modelo de datos persistente. Se utiliza una base de datos H2 (en memoria/archivo) optimizada con índices para búsquedas rápidas por Hash.

3 Diagrama de Secuencia: Detección de Mutantes

El siguiente diagrama ilustra el flujo de una petición `POST /mutant`. Se destaca la estrategia de **Caching/Deduplicación** utilizando SHA-256 para evitar el re-procesamiento costoso de ADNs ya analizados.



4 Detalles del Algoritmo y Optimizaciones

El núcleo de la detección se encuentra en la clase `MutantDetector`. Para garantizar el cumplimiento de los SLAs de rendimiento (milisegundos para matrices estándar y <1s para matrices grandes), se implementaron las siguientes optimizaciones:

4.1 1. Eficiencia Temporal: Early Termination

El algoritmo no recorre la matriz completa necesariamente. Se mantiene un contador de secuencias encontradas.

- Tan pronto como el contador supera 1 (`sequenceCount > 1`), el bucle se interrumpe inmediatamente y retorna `true`.
- Esto mejora la complejidad promedio de $O(N^2)$ a un comportamiento cercano a $O(N)$ en casos positivos.

4.2 2. Eficiencia Espacial: Matriz de Caracteres

En lugar de trabajar con el array de Strings original, se realiza una conversión inicial a `char[][]`.

- Permite acceso directo por coordenadas `[i][j]`.
- Evita la sobrecarga de llamadas repetitivas a métodos como `string.charAt()`.

4.3 3. Deduplicación por Hashing

Antes de ejecutar el algoritmo costoso, se genera un Hash SHA-256 único de la secuencia de ADN.

- Se consulta un índice B-Tree en la base de datos (`findByDnaHash`).
- Si el ADN ya fue procesado, se retorna el resultado histórico en tiempo $O(1)$ (ignorando latencia de red), evitando el costo computacional del análisis.