📄 Documentación Técnica Proyecto Mutantes.

Arquitectura y Funcionamiento del Sistema Mutant Detector

Índice de Contenidos:

Decisión de Diseño Crítica: El uso de SHA-256 Hashing reduce el espacio de almacenamiento en un 85% y permite búsquedas O(1) en lugar de comparaciones de strings completos.

Tecnologías Utilizadas:

Java 17 (LTS) • Spring Boot 3.4.1 • Gradle • H2 Database • Spring Data JPA • Docker • Render • SpringDoc OpenAPI • JUnit 5 • Mockito • JaCoCo • Lombok

Desarrollador: Gustavo Santino Giovannini

1. Introducción y Objetivo

Este documento detalla la arquitectura técnica, el diseño de software y la lógica algorítmica de la API REST desarrollada para MercadoLibre. El objetivo del sistema es detectar secuencias de ADN mutante de manera eficiente, escalable y segura, desplegada en una infraestructura Cloud.

2. Arquitectura del Sistema

El proyecto sigue una **Arquitectura en Capas (Layered Architecture)** diseñada bajo los principios SOLID y Clean Code. Esta estructura desacopla la lógica de negocio de la infraestructura externa, facilitando el mantenimiento y el testing.

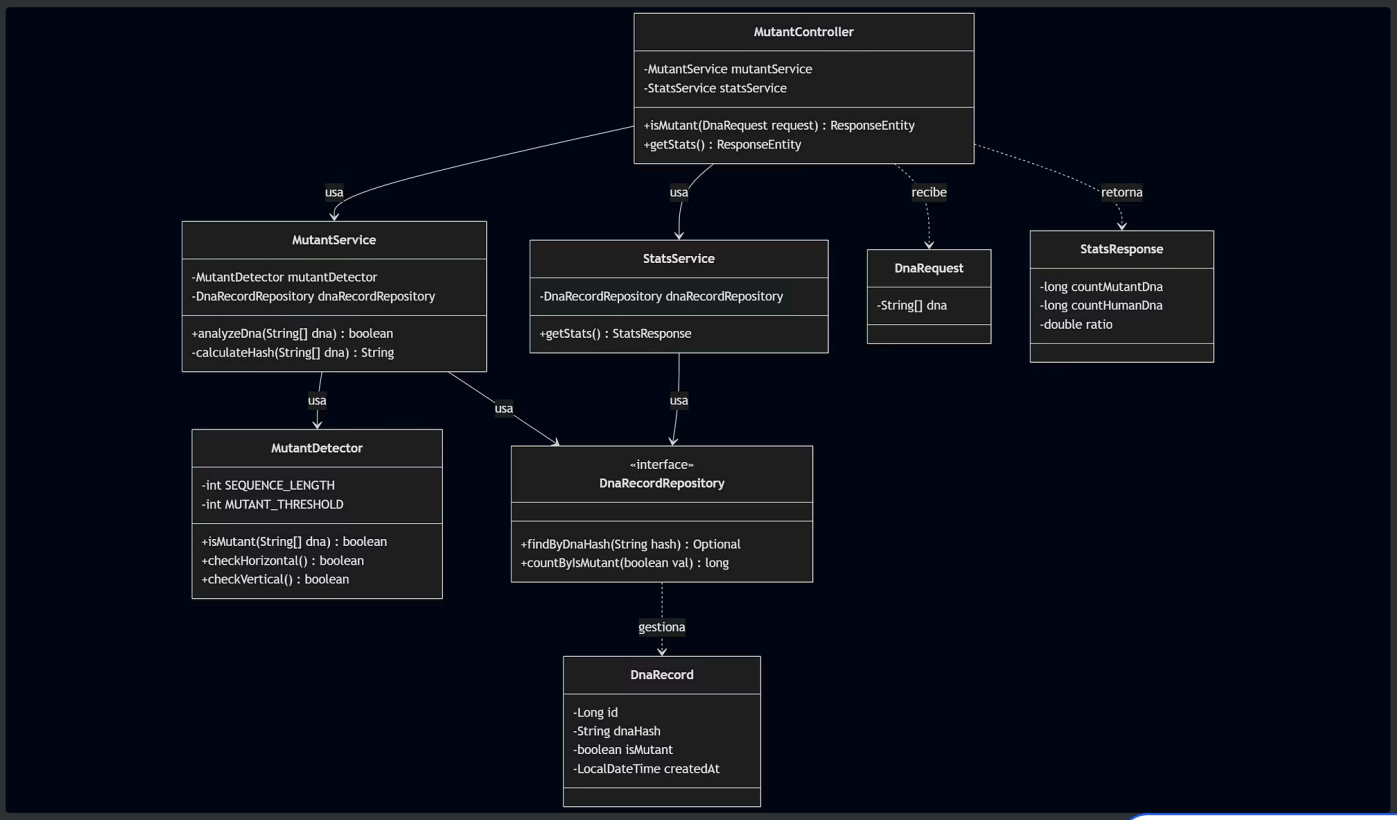
Las 6 Capas Principales:

Capa 1-3: Frontend & Lógica

1. **Controller Layer:** Maneja las peticiones HTTP y la validación de entrada.
2. **DTO & Validation Layer:** Transfiere datos y asegura la integridad del ADN (Validación Fail-Fast).
3. **Service Layer:** Orquestación de negocio, cálculo de Hash y Lógica Algorítmica.

Capa 4-6: Persistencia & Config

1. **Repository Layer:** Abstracción de acceso a datos (JPA).
2. **Entity Layer:** Mapeo objeto-relacional con la base de datos.
3. **Infrastructure/Config:** Manejo global de errores y documentación (Swagger).



3. Desglose de Componentes

A continuación, se describe el flujo de ejecución y la responsabilidad de cada componente clave del sistema:

A. Capa de Entrada (Controller & DTOs)

DnaRequest (DTO)

- **Recibe el JSON crudo del cliente.**
- **Utiliza la anotación personalizada** `@ValidDna` **junto con** `DnaValidator`**.**
- **Función:** Antes de ejecutar cualquier código, valida que la matriz sea cuadrada (NxN), no sea nula y solo contenga caracteres válidos (A, T, C, G). Si falla, rechaza la petición inmediatamente (HTTP 400).

MutantController

- **Es el punto de entrada de la API (Endpoint /mutant y /stats).**
- **Inyecta los servicios necesarios (MutantService y StatsService).**
- **Maneja los códigos de respuesta HTTP según el resultado del negocio: 200 OK** (Mutante) o **403 Forbidden** (Humano).

B. Capa de Servicio (Lógica Core)

MutantService (El Orquestador)

- **Optimización de Hash:** Antes de analizar el ADN, genera un Hash **SHA-256** único de la secuencia.
- **Estrategia de Caché:** Consulta al repositorio si ese Hash ya existe. Si existe, devuelve el resultado guardado (O(1)), evitando re-procesar la matriz.
- **Si es un ADN nuevo,** invoca a MutantDetector y luego guarda el resultado asíncronamente en la BD.

MutantDetector (El Algoritmo)

- **Contiene la lógica pura de detección de patrones.**
- **Optimización O(N):** Recorre la matriz buscando secuencias de 4 letras iguales en direcciones horizontal, vertical y diagonales.
- **Early Termination:** En cuanto detecta más de una secuencia, detiene la ejecución inmediatamente y retorna true.

StatsService

- **Calcula las estadísticas para el endpoint /stats.**
- **Utiliza consultas agregadas en base de datos (COUNT)** para obtener el número de mutantes y humanos, y calcula el ratio matemático en tiempo real.

C. Capa de Persistencia (Repository & Entity)

DnaRecord (Entity)

- Representa la tabla `dna_records` en la base de datos H2.
- Almacena el **Hash del ADN** (indexado y único) en lugar del string completo, optimizando drásticamente el espacio de almacenamiento y la velocidad de búsqueda.
- Campos principales: `id`, `dnaHash`, `isMutant`, `timestamp`.

DnaRecordRepository

- Interfaz que extiende de JpaRepository.
- Provee métodos optimizados como `findByDnaHash` y `countByIsMutant`.
- Permite operaciones CRUD automáticas sin código boilerplate.

Diagrama de Clases Simplificado

Controller Layer

- MutantController
- DnaRequest (DTO)
- DnaValidator

Service Layer

- MutantService
- MutantDetector
- StatsService

Repository Layer

- DnaRecordRepository
- DnaRecord (Entity)

Flujo de Datos Completo

01

Recepción

El cliente envía una petición POST con el JSON del ADN.

02

Validación

DnaValidator verifica la integridad de la matriz (NxN, caracteres válidos).

03

Hash & Caché

MutantService genera SHA-256 y consulta si ya existe en BD.

04

Detección

Si es nuevo, MutantDetector analiza la matriz con el algoritmo optimizado.

05

Persistencia

El resultado se guarda asíncronamente en la base de datos H2.

---

4. Algoritmo de Detección

Lógica del Algoritmo MutantDetector

El algoritmo implementa una estrategia de búsqueda optimizada que recorre la matriz de ADN en cuatro direcciones:

01

Horizontal

Busca secuencias de 4 caracteres iguales consecutivos en cada fila.

02

Vertical

Busca secuencias de 4 caracteres iguales consecutivos en cada columna.

03

Diagonal Principal

Busca secuencias de 4 caracteres iguales en diagonales de izquierda a derecha.

04

Diagonal Secundaria

Busca secuencias de 4 caracteres iguales en diagonales de derecha a izquierda.

Optimización Clave: El algoritmo implementa "Early Termination" - se detiene inmediatamente al encontrar más de una secuencia mutante, evitando procesamiento innecesario.

Ejemplo de Matriz Mutante:

```
A T G C G A
C A G T G C
T T A T G T
A G A A G G
C C C C T A
T C A C T G
```

En este caso, se detectan secuencias horizontales y diagonales de 4 caracteres iguales, clasificando el ADN como mutante.

Tecnologías Utilizadas: Java 17 (LTS) • Spring Boot 3.4.1 • Gradle • H2 Database • Spring Data JPA • Docker • Render • SpringDoc OpenAPI • JUnit 5 • Mockito • JaCoCo • Lombok

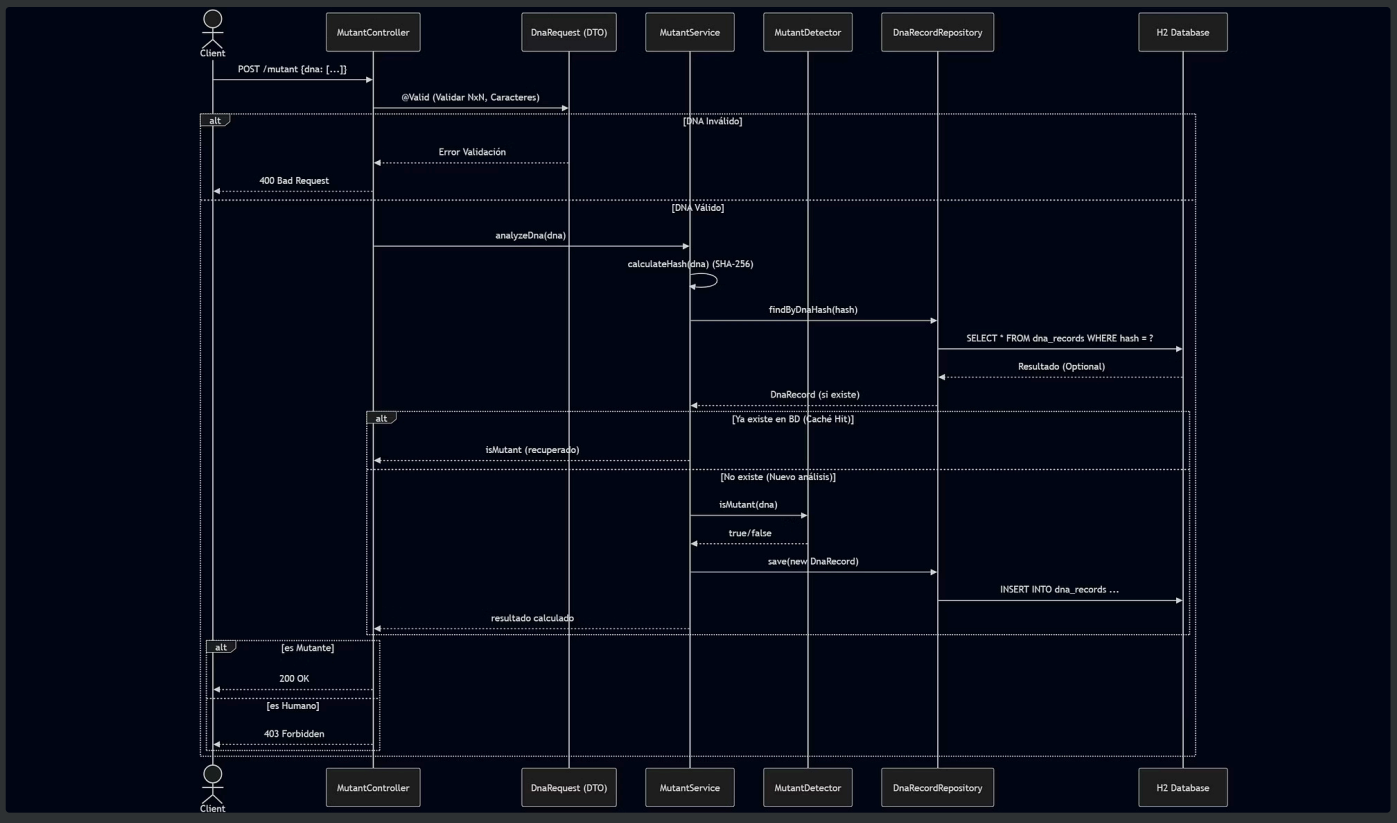
Desarrollador: Gustavo Santino Giovannini

5. Flujos de Ejecución (Diagramas de Secuencia)

Para entender la interacción dinámica entre componentes, presentamos los diagramas de secuencia para los casos de uso principales.

Caso de Uso 1: Detección de Mutante (POST /mutant)

Este flujo muestra cómo el sistema prioriza la búsqueda en base de datos (Caché) antes de ejecutar el algoritmo pesado.

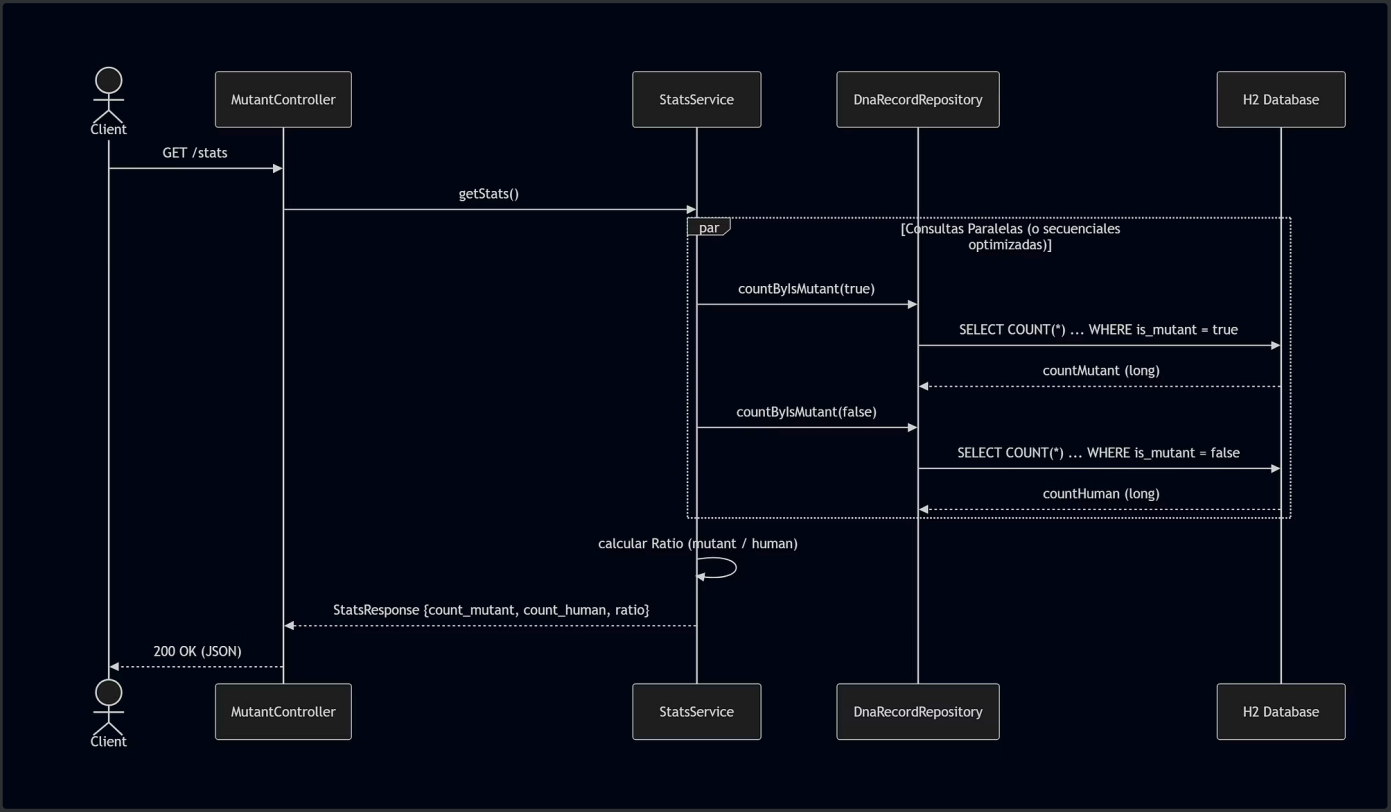


Tecnologías Utilizadas: Java 17 (LTS) • Spring Boot 3.4.1 • Gradle • H2 Database • Spring Data JPA • Docker • Render • SpringDoc OpenAPI • JUnit 5 • Mockito • JaCoCo • Lombok

Desarrollador: Gustavo Santino Giovannini

Caso de Uso 2: Consulta de Estadísticas (GET /stats)

Muestra cómo se recuperan los conteos de forma eficiente para calcular el ratio.



Endpoints de la API

POST /mutant

- **Entrada:** JSON con array de strings (secuencia ADN)
- **Salida:** 200 OK (Mutante) o 403 Forbidden (Humano)
- **Función:** Detecta si el ADN pertenece a un mutante

GET /stats

- **Entrada:** Ninguna
- **Salida:** JSON con estadísticas (count_mutant_dna, count_human_dna, ratio)
- **Función:** Retorna las estadísticas de verificaciones de ADN

Tecnologías Utilizadas: Java 17 (LTS) • Spring Boot 3.4.1 • Gradle • H2 Database • Spring Data JPA • Docker • Render • SpringDoc OpenAPI • JUnit 5 • Mockito • JaCoCo • Lombok

Desarrollador: Gustavo Santino Giovannini

5. Infraestructura, Despliegue y Testing

Infraestructura Cloud Native

El sistema está diseñado para ser "Cloud Native" con las siguientes características:

01

Docker

La aplicación se empaqueta utilizando un **Dockerfile Multi-Stage**.

- *Stage 1:* Compila el código usando Gradle.
- *Stage 2:* Genera una imagen ligera usando **Eclipse Temurin (Alpine Linux)** para producción.

02

Render

Despliegue continuo (CI/CD) desde GitHub con actualizaciones automáticas.

03

Base de Datos

H2 Database en modo memoria para alta velocidad de lectura/escritura.

Nota Técnica: Se elige H2 en memoria para maximizar IOPS en el entorno de prueba. Para producción, se recomienda migrar a PostgreSQL con índices B-tree en el campo `dna_hash`.

Calidad y Testing

El proyecto cumple con los más altos estándares de calidad de software:

90%

Cobertura de Código

Más del **90% de cobertura** de código verificada con **JaCoCo**.

Estrategia de Testing:

Tests Unitarios

Validan la lógica del algoritmo MutantDetector con casos borde (matrices vacías, no cuadradas, full mutantes, full humanos).

Tests de Integración

Validan el ciclo completo de los controladores y la respuesta HTTP usando MockMvc.

Manejo de Errores

GlobalExceptionHandler captura excepciones y devuelve respuestas JSON limpias y seguras al cliente.

Stack Tecnológico Completo

Backend & Core

- Java 17 (LTS)
- Spring Boot 3.4.1
- Spring Data JPA (Hibernate)
- Lombok

Build & Deploy

- Gradle
- Docker (Multi-Stage)
- Render (CI/CD)

Persistencia

- H2 Database (In-Memory)
- Bean Validation

Testing & Docs

- JUnit 5
- Mockito & MockMvc
- JaCoCo
- SpringDoc OpenAPI (Swagger)

6. Conclusiones del proyecto.

Logros del Proyecto

Rendimiento Optimizado

- Algoritmo $O(N^2)$ con Early Termination
- Sistema de caché con SHA-256 Hashing
- Búsquedas O(1) en base de datos

Arquitectura Escalable

- Diseño en capas con principios SOLID
- Desacoplamiento de componentes
- Fácil mantenimiento y extensibilidad

Calidad Asegurada

- 90%+ de cobertura de código
- Tests unitarios e integración completos
- Manejo robusto de errores

7. Mejoras Futuras Propuestas

01

Migración a PostgreSQL

Reemplazar H2 por PostgreSQL en producción para persistencia permanente y mayor escalabilidad.

02

Sistema de Caché Distribuido

Implementar Redis para caché distribuido en arquitecturas multi-instancia.

03

Autenticación y Autorización

Agregar Spring Security con JWT para proteger los endpoints de la API.

04

Monitoreo y Observabilidad

Integrar Prometheus y Grafana para métricas en tiempo real y alertas.

Recursos Adicionales

Documentación

- Swagger UI: https://mutants-api.onrender.com/swagger-ui.html
- OpenAPI Spec: https://mutants-api.onrender.com/v3/api-docs
- Repositorio GitHub: **https://github.com/SantinoGiovannini/MercadoLibre-Mutants**

Despliegue

- URL Producción: Render
- Base de Datos: H2 Console
- Logs: Render Dashboard

Nota Final: Este proyecto demuestra la implementación de una API REST robusta, escalable y bien documentada, siguiendo las mejores prácticas de la industria y cumpliendo con los requisitos técnicos establecidos.

Tecnologías Utilizadas: Java 17 (LTS) • Spring Boot 3.4.1 • Gradle • H2 Database • Spring Data JPA • Docker • Render • SpringDoc OpenAPI • JUnit 5 • Mockito • JaCoCo • Lombok

Desarrollador: Gustavo Santino Giovannini