Informe Técnico: Mutant Detector API

Examen Final - Mercadolibre

- Autor: Enzo Fernández
- Fecha: 22/11/2025
- Proyecto: Mutant Detector API

Tabla de Contenidos

---

1. Introducción

Este documento detalla la implementación de la "Mutant Detector API", un proyecto desarrollado para el desafío técnico de Mercadolibre. El objetivo principal es detectar si un humano es mutante basándose en su secuencia de ADN, siguiendo la solicitud de Magneto para reclutar miembros para su causa.

El proyecto cumple y supera los tres niveles del desafío, implementando:

- Un algoritmo de detección eficiente.
- Una API REST completamente funcional desplegada en la nube.
- Un sistema de persistencia con base de datos H2 para almacenar y consultar los resultados, incluyendo un endpoint de estadísticas.

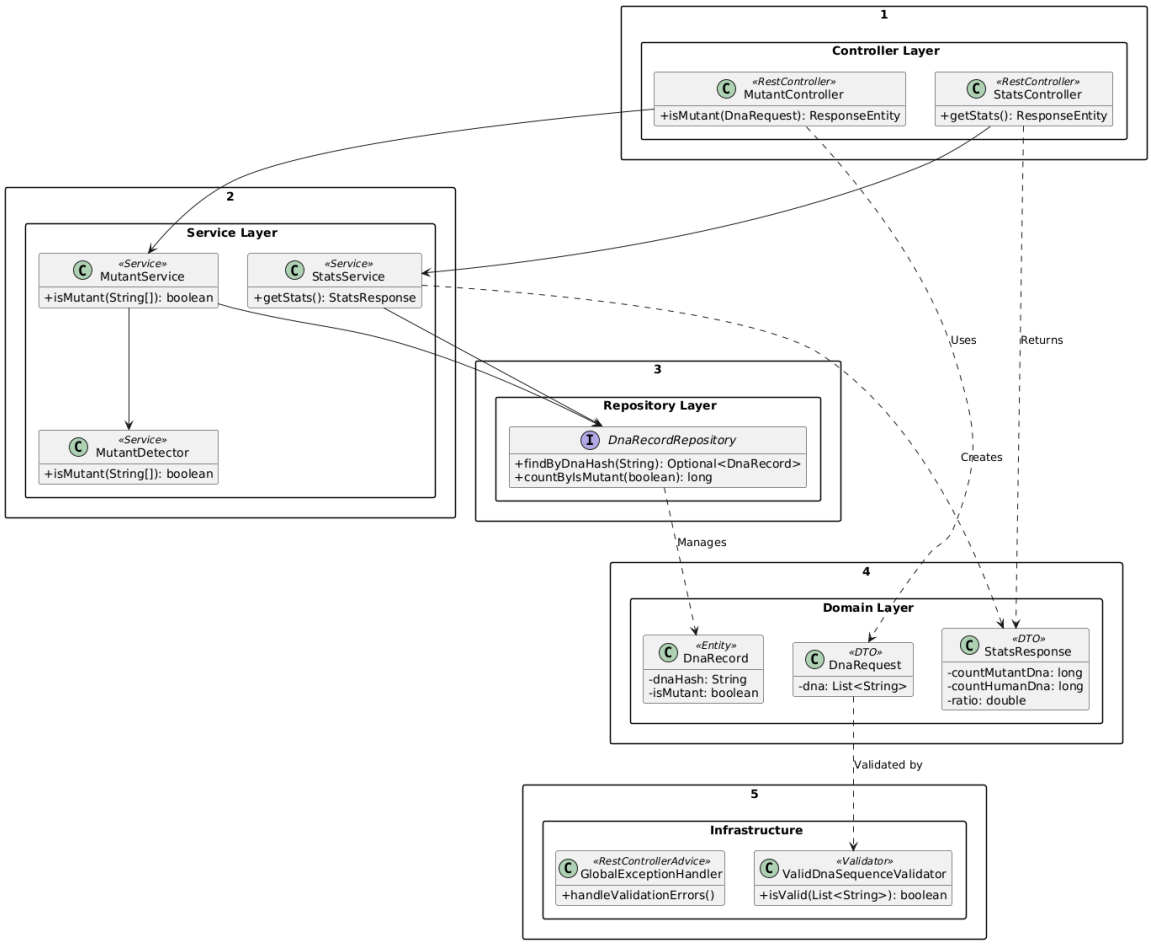
---

2. Arquitectura del Proyecto

La aplicación se ha construido siguiendo una arquitectura de 6 capas, un patrón estándar en Spring Boot que promueve la separación de responsabilidades, la mantenibilidad y la testabilidad del código.

- controller: Expone los endpoints REST (/mutant, /stats) y maneja las peticiones y respuestas HTTP.
- service: Contiene la lógica de negocio principal. Orquesta las llamadas al detector, al repositorio y calcula las estadísticas.
- repository: Define la interfaz para el acceso a datos, extendiendo JpaRepository para interactuar con la base de datos.
- entity: Contiene las clases que representan las tablas de la base de datos (entidades JPA).
- dto: Data Transfer Objects. Define los contratos de datos para las peticiones y respuestas de la API.
- exception: Implementa el manejo de excepciones global para toda la aplicación.
- validation: Contiene las validaciones personalizadas para los datos de entrada.

Diagrama de Clases



3. **Algoritmo de Detección (isMutant)**

El núcleo del proyecto es el algoritmo que determina si una secuencia de ADN es mutante. Un ADN es considerado mutante si contiene más de una secuencia de cuatro letras idénticas en dirección horizontal, vertical o diagonal.

Lógica y Validaciones

- Validación de Entrada: Antes del análisis, se valida que el ADN sea una matriz NxN (con N >= 4) y que solo contenga los caracteres permitidos (A, T, C, G), utilizando un validador personalizado (@ValidDnaSequence).
- Búsqueda de Secuencias: El algoritmo recorre la matriz una sola vez, buscando secuencias en las cuatro direcciones posibles (horizontal, vertical, diagonal principal y diagonal inversa) desde cada celda.

Optimizaciones Implementadas

- Early Termination (Terminación Anticipada): La búsqueda se detiene y retorna true inmediatamente en cuanto el contador de secuencias encontradas supera 1. Esto evita procesar la matriz completa innecesariamente en la mayoría de los casos mutantes.
- Conversión a char[][]: El String[] de entrada se convierte a una matriz de caracteres (char[][]) una sola vez al inicio, permitiendo un acceso O(1) a cada base de ADN.
- Boundary Checking (Comprobación de Límites): Se verifica si hay espacio suficiente para una secuencia de 4 letras antes de intentar la búsqueda, evitando accesos fuera de los límites de la matriz.
- Comparaciones Directas: La comprobación de secuencias se realiza mediante comparaciones booleanas directas (&&) en lugar de bucles internos, lo que resulta más rápido.Complejidad

Complejidad

- Complejidad Temporal: $O(N^2)$ en el peor de los casos (un ADN humano), pero gracias al early termination, el rendimiento práctico es cercano a O(N) para ADN mutante.
- Complejidad Espacial: O(1) de espacio adicional, ya que solo se utiliza un contador. La matriz char[][] es una conversión necesaria, no una estructura auxiliar.

---

4. API REST

La funcionalidad se expone a través de dos endpoints REST.

POST /mutant: Verifica si una secuencia de ADN pertenece a un mutante.

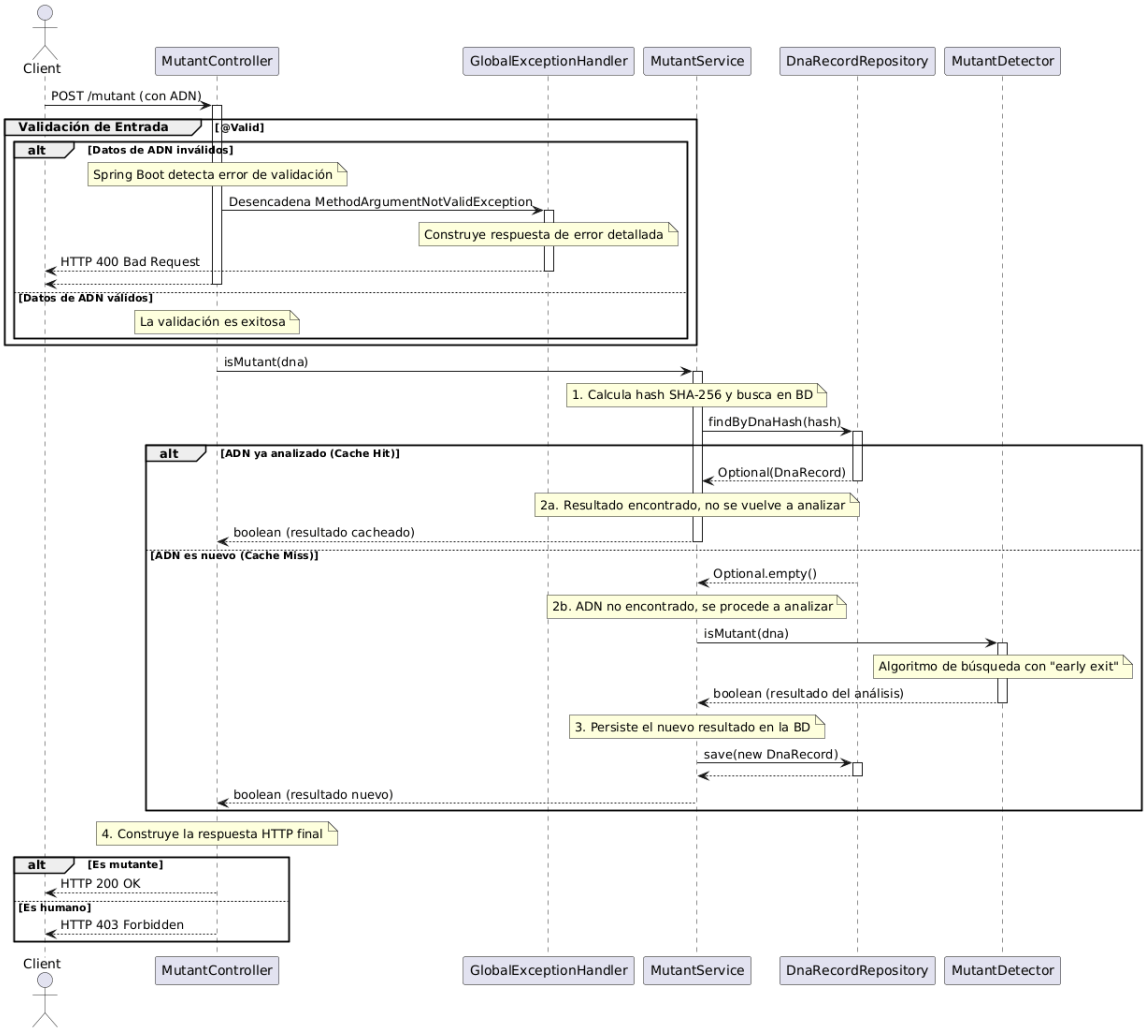
Body (JSON):

```
{
  "dna": ["ATGCGA","CAGTGC","TTATGT","AGAAGG","CCCCTA","TCACTG"]
}
```

Respuestas:

- 200 OK: Si el ADN es mutante.
- 403 Forbidden: Si el ADN es humano.•
- 400 Bad Request: Si el ADN es inválido (no es NxN, contiene caracteres no permitidos, etc.).

Diagrama de Secuencia: POST /mutant

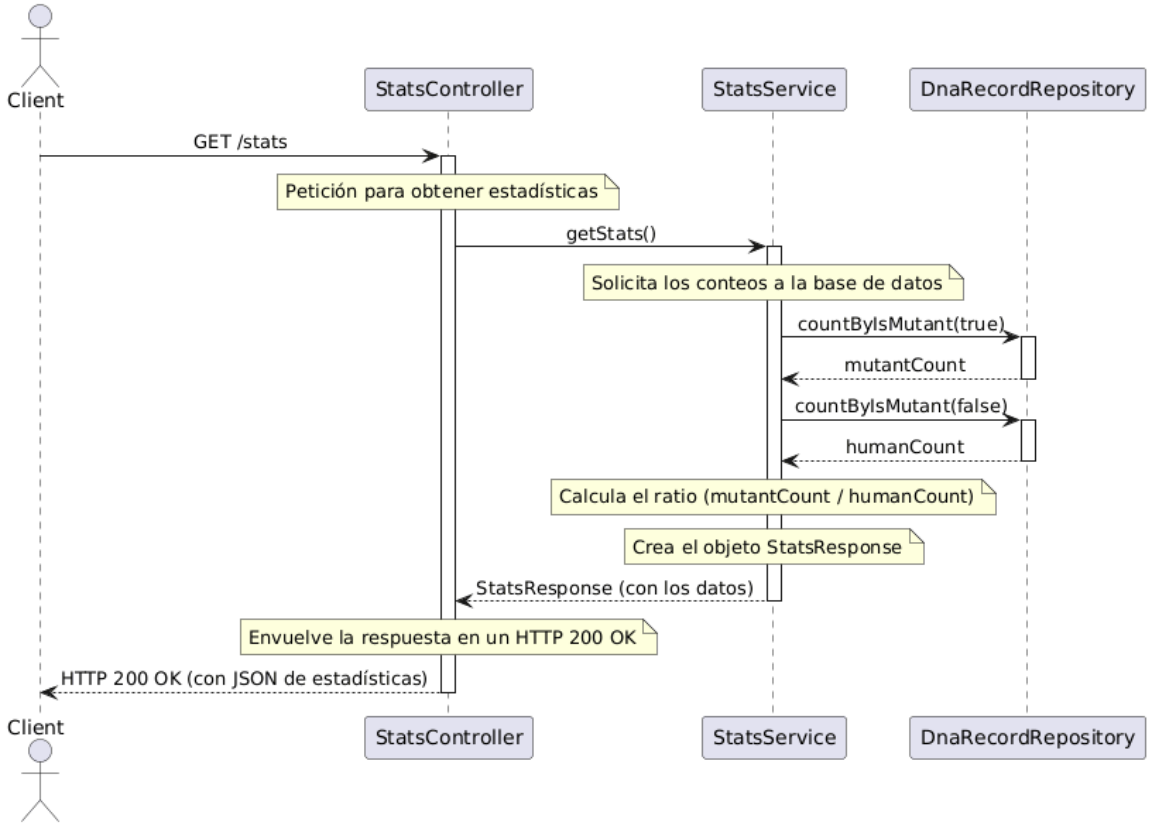


GET /stats: Devuelve las estadísticas de las verificaciones de ADN realizadas.

Respuesta (JSON):


```
{
  "count_mutant_dna": 40,
  "count_human_dna": 100,
  "ratio": 0.4
}
```


Diagrama de Secuencia: POST /mutant



5. **Persistencia de Datos**

   Para cumplir con el Nivel 3, se implementó un sistema de persistencia utilizando Spring Data JPA y una base de datos en memoria H2.
   Estrategia de Deduplicación
   Para garantizar que solo se guarde un registro por ADN, se implementó una estrategia de hash:

   1. Antes de analizar un ADN, se calcula su hash SHA-256.
   2. Se consulta la base de datos para ver si ya existe un registro con ese hash.
   3. Si existe, se devuelve el resultado almacenado sin volver a analizarlo.
   4. Si no existe, se analiza el ADN, se guarda el resultado junto con su hash en la base de datos, y luego se devuelve.

   La entidad DnaRecord utiliza una restricción unique=true en la columna del hash para garantizar la unicidad a nivel de base de datos.

---

6. **Manejo de Errores y Validaciones**

   La aplicación cuenta con un sistema robusto de manejo de errores centralizado en la clase GlobalExceptionHandler (anotada con @RestControllerAdvice). Este manejador captura excepciones específicas como:

   - MethodArgumentNotValidException: Para errores de validación en los DTOs.
   - IllegalArgumentException: Para errores de lógica de negocio (ej. ADN inválido detectado en la capa de servicio).
   - DnaHashCalculationException: Para errores internos durante el cálculo del hash.

---

7. **Testing y Cobertura de Código**

   La calidad y correctitud del código se aseguran mediante una suite de tests exhaustiva utilizando JUnit 5 y Mockito.

   - Tests Unitarios: Cubren la lógica del algoritmo en MutantDetector y la lógica de negocio en los servicios.
   - Tests de Integración: Verifican el comportamiento de los endpoints del controlador.
   - Cobertura de Código: El proyecto alcanza una cobertura total del 92%, superando el requisito del 80%.

---

8. **Despliegue en Producción**
   La API ha sido desplegada en Render, una plataforma de cloud computing libre. El despliegue se realiza a través de un Dockerfile multi-stage que optimiza el tamaño de la imagen final.
   - URL de la API Base: https://mutant-detector-hiin.onrender.com/
   - URL de Swagger UI: https://mutant-detector-hiin.onrender.com/swagger-ui/index.html

---

9. **Conclusiones**
   El proyecto "Mutant Detector API" cumple con todos los requisitos del desafío propuesto. Se ha desarrollado una solución robusta, eficiente y bien documentada que sigue las mejores prácticas de la industria para el desarrollo con Spring Boot. La arquitectura modular, las optimizaciones del algoritmo y la alta cobertura de tests garantizan un producto de alta calidad.

---

10. **Mejoras Futuras**
    - Migración de Base de Datos: Cambiar de H2 a una base de datos más robusta como PostgreSQL para un entorno de producción real.
    - Caché: Implementar una capa de caché con Redis para las estadísticas y los resultados de ADN, reduciendo la carga sobre la base de datos.
    - Seguridad: Añadir autenticación (ej. JWT) a los endpoints si fuera necesario.
    - Métricas y Monitoreo: Integrar Prometheus y Grafana para monitorear el rendimiento y la salud de la aplicación.