Título del Proyecto: Proyecto Mutantes

Descripción General: El Proyecto Mutantes es una aplicación diseñada para detectar secuencias de ADN que pertenecen a individuos mutantes. Utiliza un patrón de búsqueda de trie para analizar las secuencias y determinar si un individuo presenta características genéticas inusuales (Secuencias de cuatro caracteres iguales en forma vertical, horizontal o diagonal).

Funcionalidades Principales

1. **Recepción de Secuencias de ADN:**

   Permite la entrada de secuencias de ADN en formato de matriz, donde cada fila representa una cadena de nucleótidos.

2. **Validación de Datos:**

   Verifica que las secuencias contengan solo caracteres válidos (A, T, C, G).

   Asegura que la matriz no esté vacía y que tenga un formato correcto.

3. **Detección de Mutantes:**

   Implementa un algoritmo que busca combinaciones de cuatro nucleótidos idénticos en diferentes orientaciones (horizontal, vertical y diagonal).

   Clasifica a los individuos como mutantes si se detectan las secuencias específicas.

4. **Interfaz de Usuario:**

   Proporciona una API REST que permite a los usuarios enviar secuencias de ADN y recibir respuestas sobre la clasificación.

5. **Almacenamiento de Resultados:**

   Los resultados de las verificaciones se pueden almacenar en una base de datos para análisis futuros.

Descripción del problema:

En este proyecto se pidió al alumno el realizar una API Rest que detectara mutantes en base a la composición de una tabla de tamaño NxN que únicamente puede contener las letras A,T,C,G.

Sabremos si un humano es mutante, si encuentras más de una secuencia de cuatro letras iguales, de forma oblicua, horizontal o vertical.

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

No-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

Mutante

Creí importante para el desarrollo de este proyecto no solo determinar qué arreglos ingresados nos da como resultado un mutante, sino también su persistencia en la base de datos, por lo cual hago uso de la entidad mutante, la cual tiene parámetros tales como un id, nombre de la persona, apellido y su cadena de ADN.

Para la correcta implementación del proyecto se utilizó una arquitectura en tres capas (Controladores, servicios y repositorios) para las cuales usé como base mi clase más importante “Mutante”.

Dentro de la carpeta main/java encontramos la carpeta contenedora principal com.example.mutantes que contiene todos los archivos usados para el proyecto:

- **Carpeta config:** esta carpeta contiene el archivo “CustomRevisionListener” que es una implementación de un listener de revisiones en Hibernate Envers, que permite manejar eventos relacionados con la creación de nuevas revisiones de auditoría.
- **Carpeta controllers:** aquí se guardan los controladores que manejarán las solicitudes HTTP. En este caso solo tiene al archivo “MutanteController” que es un controlador REST que maneja las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) y otras funcionalidades relacionadas con la entidad Mutante. Utiliza el servicio MutanteService para realizar la lógica de negocio y devuelve respuestas adecuadas a las solicitudes HTTP. Las operaciones incluyen obtener todos los mutantes, obtener un mutante por ID, actualizar un mutante, eliminar un mutante, analizar ADN para determinar si una persona es un mutante y obtener estadísticas sobre los mutantes.
- **Carpeta entities:** En esta carpeta se guardan las entidades relacionadas directamente al funcionamiento del negocio(Mutante), y además incluye dos subcarpetas:
    - **audit:** que contiene la entidad revisión, la cual se encarga de almacenar información sobre las revisiones realizadas en otras entidades de la aplicación. Cuando se realizan cambios en las entidades que están siendo auditadas, Hibernate Envers crea una nueva entrada en la tabla de revisiones, utilizando esta entidad para registrar el número de revisión y la marca de tiempo correspondiente.
    - **dtos:** contiene la entidad EstadisticasDTO. Esta entidad es parte de la lógica de negocio, encargada de llevar la cuenta de la cantidad de mutantes, humanos y su proporción.
- **Carpeta repositories:** guarda a “MutanteRepository” que es un repositorio que permite realizar operaciones de acceso a datos sobre la entidad Mutante. Proporciona

un método personalizado para contar el número de mutantes en función de su estado (esMutante), además de heredar métodos estándar para operaciones CRUD gracias a la extensión de JpaRepository. Esto facilita la interacción con la base de datos y mejora la mantenibilidad del código.

- **Carpeta search:** aquí encontramos las entidades necesarias para la lógica de la búsqueda de patrones con trie, en donde se encuentran las entidades Arbol y Nodo(se explica la lógica más adelante.)
- **Carpeta services:** en esta carpeta están las entidades que contienen toda la lógica del negocio. Más adelante está explicada su funcionalidad.

Búsqueda en trie:

El Trie es una estructura de datos especializada que se utiliza para almacenar cadenas de texto, donde cada nodo representa un carácter de la cadena. La operación de búsqueda en un Trie permite verificar de manera eficiente si una palabra existe en la estructura.

1- Estructura de TrieNode:

Cada nodo en el Trie contiene:

- Un arreglo (o mapa) de nodos hijos, donde cada índice corresponde a un carácter (por ejemplo, para letras en inglés en minúscula, un arreglo de tamaño 26).
- Una bandera booleana que indica si una palabra termina en ese nodo.

```
public class Nodo {
    Nodo[] hijos;
    boolean esFinDePalabra;

    public Nodo() {
        hijos = new Nodo[4]; // Para A, C, G, T
        esFinDePalabra = false;
    }
}
```

2- Estructura del árbol:

- **Atributos:**
  - raiz: Es el nodo raíz del Trie. Se inicializa en el constructor de la clase.
  - adn: Un arreglo de cadenas que puede ser utilizado para almacenar secuencias de ADN.
- **Constructor:** El constructor inicializa el nodo raíz del Trie. Se asume que la clase Nodo es otra clase que representa un nodo en el Trie y que tiene un arreglo de nodos hijos.
- **Método insertar:**

```
public void insertar(String palabra) {
    palabra = palabra.toUpperCase(); // Convertir a mayúsculas
    Nodo nodo = raiz;
    for (char c : palabra.toCharArray()) {
        int indice = charAIndice(c);
        if (nodo.hijos[indice] == null) {
            nodo.hijos[indice] = new Nodo();
```

```
            }
            nodo = nodo.hijos[indice];
        }
        nodo.esFinDePalabra = true;
    }
```

- Este método permite insertar una palabra (o secuencia de ADN) en el Trie.
- Convierte la palabra a mayúsculas para asegurar que la búsqueda sea insensible a mayúsculas y minúsculas.
- Itera sobre cada carácter de la palabra:
- Convierte el carácter a un índice utilizando el método charAIndice.
- Si el nodo hijo correspondiente al índice es null, crea un nuevo nodo.
- Mueve el puntero al nodo hijo correspondiente.
- Al final, marca el nodo como el final de una palabra (esFinDePalabra = true).

- **Método buscar:**

```
public boolean buscar(String palabra) {
    palabra = palabra.toUpperCase(); // Convertir a mayúsculas
    Nodo nodo = raiz;
    for (char c : palabra.toCharArray()) {
        int indice = charAIndice(c);
        if (nodo.hijos[indice] == null) {
            return false;
        }
        nodo = nodo.hijos[indice];
    }
    return nodo.esFinDePalabra;
}
```

- Este método permite buscar una palabra en el Trie.
- Al igual que en el método insertar, convierte la palabra a mayúsculas.
- Itera sobre cada carácter de la palabra:
  - Convierte el carácter a un índice utilizando el método charAIndice.
  - Si el nodo hijo correspondiente al índice es null, devuelve false (la palabra no existe).
  - Si existe, mueve el puntero al nodo hijo correspondiente.
- Al final, devuelve true si el nodo actual marca el final de una palabra, o false si no.

- **Método charAIndice:** Este método convierte un carácter de ADN ('A', 'C', 'G', 'T') en un índice correspondiente (0, 1, 2, 3). Si se pasa un carácter inválido, lanza una excepción.

3- Lógica

Se encuentra implementada en mutanteService, donde se genera un árbol y la longitud de la secuencia a buscar

```
 private Arbol arbol;
private static final int LONGITUD_SECUENCIA = 4;

public MutanteService() {
    this.arbol = new Arbol();
    inicializarArbol();
}

private void inicializarArbol() {
    // Secuencias de ADN que se consideran mutantes
    arbol.insertar("AAAA");
```

```
    arbol.insertar("TTTT");
    arbol.insertar("CCCC");
    arbol.insertar("GGGG");
}
```

Además se insertan las secuencias a buscar.

En siguiente encontramos el método analizarADN que se encarga de analizar la secuencia de ADN de un objeto de tipo Mutante. Este método determina si la secuencia de ADN pertenece a un "mutante" y realiza algunas acciones adicionales basadas en el resultado. A continuación, te explico en detalle cómo funciona este método.

```
public boolean analizarADN(Mutante persona) {
    if (persona.getAdn() == null || persona.getAdn().length == 0) {
        throw new IllegalArgumentException("Secuencia de ADN
inválida");
    }

    boolean resultado = isMutant(persona.getAdn());
    persona.setEsMutante(resultado);
    mutanteRepository.save(persona);
    return resultado;
}
```

por ultimo tenemos :

```
private boolean isMutant(String[] adn)
```

Determina si una secuencia de ADN (representada como un arreglo de cadenas) pertenece a un "mutante" según ciertas reglas. En este contexto, un "mutante" se define como una secuencia que contiene al menos dos secuencias específicas de longitud LONGITUD_SECUENCIA que se pueden encontrar en un Trie (almacenadas en la variable arbol). Utiliza una matriz booleana para evitar volver a analizar posiciones ya revisadas. La implementación es eficiente al evitar iteraciones innecesarias gracias al marcado de posiciones visitadas.

```
int n = adn.length;
int secuenciasEncontradas = 0;

// Matriz para rastrear posiciones visitadas
boolean[][] visitado = new boolean[n][n];

// Verificar horizontal y vertical en un solo bucle
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // Si la posición ya ha sido visitada, continuar
        if (visitado[i][j]) continue;
```

Usamos dos bucles para recorrer el arreglo, y en caso de que ya se haya analizado el casllero sigue con otro para evitar evaluaciones dobles

En consiguiente se verifican los caracteres de forma vertical, horizontal y diagonal (normal e inversa)

Obtener Estadísticas:

En este proceso intervienen dos entidades:

- EstadisticasDTO: aquí se guarda la información relevante(cantHumanos, cantMutantes, ratio)

- MutanteService: contiene el método con la lógica para obtener las estadísticas

```
public EstadisticasDTO obtenerEstadisticas () {
    long totalMutantes = mutanteRepository.countByEsMutante(true);
    long totalHumanos = mutanteRepository.countByEsMutante(false);
    double ratio = totalHumanos > 0 ? (double) totalMutantes /
totalHumanos : 0.0;

    return new EstadisticasDTO(totalMutantes, totalHumanos, ratio);
}
```

Diagrama de secuencia Estadísticas:

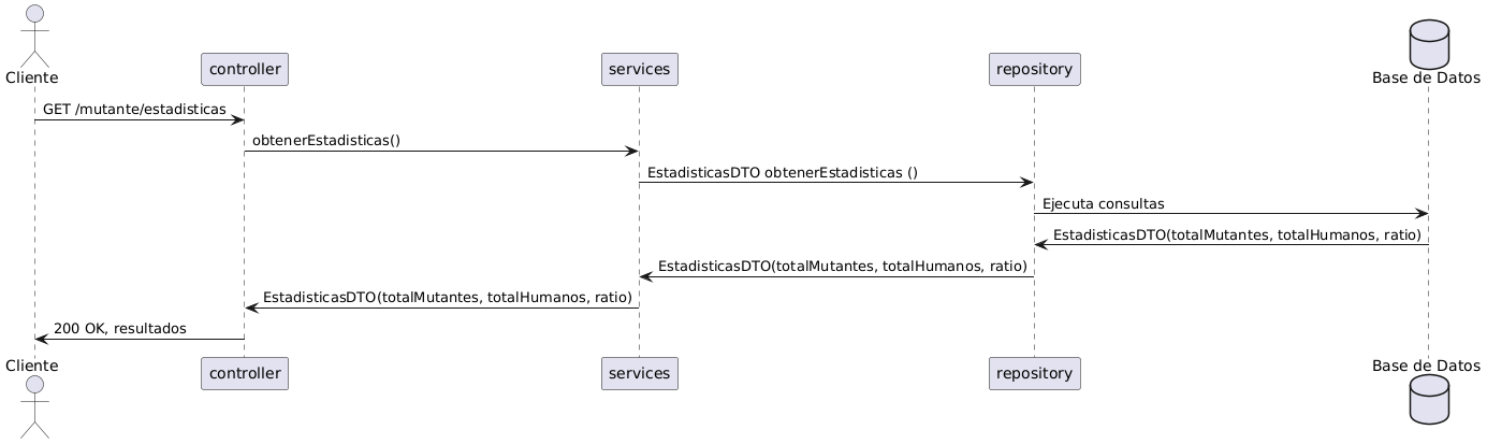

Diagrama de secuencia AnalizarMutante: